

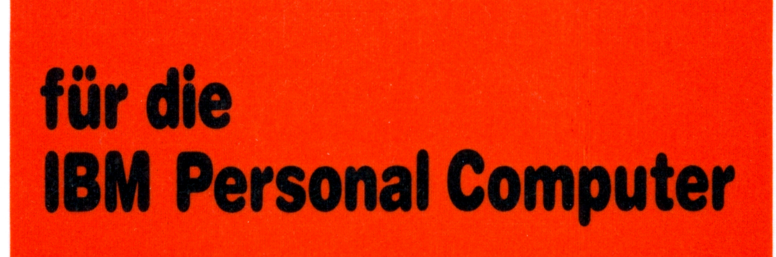
PC professionell
Arbeitsplatzrechner in Ausbildung und Praxis



David J. Bradley

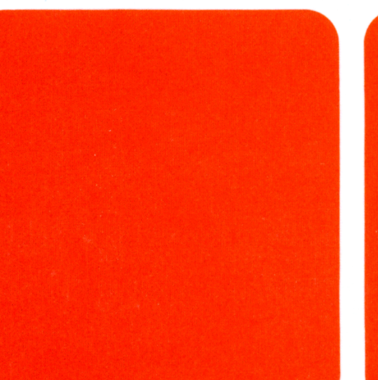


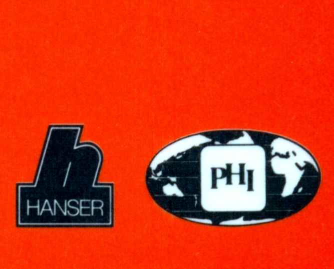
Programmieren in Assembler



für die IBM Personal Computer







HANSER
PHI

David J. Bradley

Programmieren in Assembler für die IBM Personal Computer



Herausgeber:
Prof. Dr.-Ing. Werner Heinzel, Fulda

PC professionell zeigt dem Anwender von Personal Computern den Aufbau, die Programmierung und die verschiedenen Anwendungsmöglichkeiten seines „persönlichen Rechners am Arbeitsplatz“ und führt ihn am Beispiel aktueller PC-Systeme oder mit der Darstellung und Diskussion von Fallbeispielen in die betriebliche Praxis derartiger Geräte ein.

Personalcomputer werden in den verschiedensten Anwendungsbereichen eingesetzt:

- in Klein- und Mittelbetrieben
- als persönliches Arbeitsmittel in Fachabteilungen von Großbetrieben
- als Ausbildungsmittel in Schulen, Hochschulen und Fortbildungsstätten
- im privaten Bereich

PC professionell wendet sich an diese verschiedenen Benutzer des Personal Computers.

Aufgrund des didaktischen Aufbaus und der eingearbeiteten Praxisbeispiele eignen sich die Bände zum Selbststudium.

Die Fachbuchreihe weist die folgenden thematischen Schwerpunkte auf:

Grundlagen der Hard- und Softwaretechnologie von Personal Computern,
zum Beispiel

- Systemstrukturen
- Betriebssysteme
- Programmiersprachen
- Softwarewerkzeuge

Realisierte Systeme und deren Anwendungen,
zum Beispiel

- Benutzeranleitung für bestimmte PC-Systeme
- Programmierung typischer Systeme
- Anwendungen, Fallbeispiele, Softwaresammlungen
- Softwarelösungen für bestimmte Branchen bzw. Anwendungsgebiete

Carl Hanser Verlag München Wien

David J. Bradley

Programmieren in Assembler für die IBM Personal Computer

Aus dem Amerikanischen
von Rudolf Sauerer





Eine Coedition der Verlage Carl Hanser und Prentice-Hall International

Titel der amerikanischen Originalausgabe:
Assembly Language Programming for the IBM Personal Computer by David J. Bradley

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Bradley, David J.:
Programmieren in Assembler für die IBM-Personal-Computer / David J. Bradley. Aus d. Amerikan. von Rudolf Sauerer. - München ; Wien : Hanser ; London : Prentice-Hall International, 1986.
(PC professionell)
Engl. Ausg. u. d. T.: Bradley, David J.:
Assembly language programming for the IBM personal computer
ISBN 3-446-14275-4 (Hanser)
ISBN 0-13-048984-0 (Prentice-Hall)

Eine Coedition der Verlage:
Carl Hanser Verlag München Wien
Prentice-Hall International Inc., London

Satz: Gruber + Hueber, Neutraubling
Druck: Druckerei Appl, Wemding
Buchbinderische Verarbeitung: Sellier, Freising
Umschlagentwurf: Carl-Alfred Loipersberger

Printed in Germany

Vorwort zur deutschen Ausgabe

Dieses Buch ist für einen Leserkreis gedacht, der bereits einige Erfahrung im Umgang mit Computern sammeln konnte. Grundkenntnisse in einer Programmiersprache sind also von Vorteil, ebenso wie eine gewisse Vertrautheit mit den wichtigsten Fachausdrücken, wenn sich der Anwender in die neue Materie der Programmierung in Assembler einarbeiten will. Mittels dieser Einführung werden sich ihm dann auch Details der Hardware des IBM PC erschließen, auf den dieses Buch speziell zugeschnitten ist. Anhand von teilweise lauffähigen Programmbeispielen, die die Vielfalt der Anwendungsmöglichkeiten der Assemblersprache aufzeigen sollen, wird dem Leser der Umgang mit den zur Programmierung nötigen Werkzeugen wie auch dem Betriebssystem vermittelt.

Dieses Buch ersetzt jedoch nicht eine Beschreibung der Assemblersprache oder ein Programmierhandbuch; auch ist es keine technische Beschreibung der Hardware des IBM PC.

Alle Arbeiten zur Übersetzung des vorliegenden Buches ins Deutsche wurden übrigens auf und mit dem IBM PC durchgeführt.

An dieser Stelle möchte der Übersetzer allen danken, die ihm seine Arbeit erleichterten. Dabei seien besonders erwähnt die Damen des ECO-Instituts, die die Texterfassung durchführten, und die Firma Gruber+Hueber, die den Datentransfer und den Satz des Buches besorgte.

Passau, Oktober 1985

Rudolf J. Sauerer

Vorwort

Dieses Buch soll Sie lehren, Programme in Assemblersprache auf dem IBM Personal Computer zu schreiben. Es wird Ihnen dabei auch erläutern, wie Sie Ihren IBM PC zum Schreiben dieser Programme verwenden. Und schließlich wird es Ihnen noch zeigen, wie Sie Assemblerprogramme in Verbindung mit dem Betriebssystem anwenden können.

Dieses Buch ist hauptsächlich zum Einstieg in die Assemblerprogrammierung gedacht. Programmierkenntnisse z.B. in einer höhreren Programmiersprache, werden vorausgesetzt. Deshalb wird auch nicht besonders auf Algorithmen und Programmiertechniken eingegangen. Es könnte für Sie also Probleme beim Arbeiten mit diesem Buch geben, wenn Sie nicht bereits einige Erfahrung im Schreiben von BASIC- oder Pascal-Programmen haben. Jedoch ist die Materie in einer Form dargestellt, die Ihnen den Einstieg in die Assemblerprogrammierung auch ohne Kenntnis der Arbeitsweise eines Computers ermöglichen sollte.

Der erste Abschnitt, bestehend aus den Kapiteln 1 und 2, enthält die Grundlagen der Arbeitsweise eines Computers. Der Nachdruck liegt dabei auf den Funktionen eines Rechners, die bei Benutzung einer höheren Programmiersprache nicht sofort klar erkennbar sind. Dies beinhaltet auch eine kurze Besprechung von binärer Arithmetik und interner Datendarstellung. Außerdem wird in diesem Abschnitt die Arbeitsweise des Assemblers erläutert. Haben Sie also bereits Erfahrung in Assemblerprogrammierung, so können Sie diesen Abschnitt getrost übergehen.

Der zweite Abschnitt, bestehend aus den Kapiteln 3, 4 und 7, ist eine Beschreibung und Erläuterung des Intel 8088, also des Prozessors, der im IBM PC verwendet wird. Dabei werden der 8088, seine Register und seine Adressierungsarten beschrieben. Es folgt eine Darstellung des Befehlssatzes des 8088 zusammen mit Beispielen für die meisten der beschriebenen Befehle. Kapitel 7 ist ganz dem Arithmetikprozessor 8087 gewidmet. Es beschreibt die dadurch verfügbaren zusätzlichen Datentypen und Befehle. An einigen Beispielen wird außerdem die Arbeitsweise des 8087 und der Umgang mit ihm näher erläutert.

War der vorausgehende Abschnitt breit genug gehalten, um beinahe alle Systeme abzudecken, die mit Rechnern der Familie 8086/8088 bestückt sind, so behandeln die restlichen Abschnitte dieses Buches direkt den IBM PC. Kapitel 5 und 6 erläutern die Erstellung und Anwendung von Programmen auf dem IBM Rechner. Sie werden dabei auch den Umgang mit einigen „Programmierwerkzeugen" der Assemblersprache erlernen. Dazu gehören der Zeileneditor, der Assembler und der Linker ebenso wie ein Überblick über die Arbeitsweise des Disk Operating Systems (DOS). Der Abschnitt beschreibt DOS als die Umgebung, in der Ihr Programm abläuft. Kapitel 6 behandelt zudem einige der besonderen „Programmierwerkzeuge", die als Teil des Makro-Assemblers zur Verfügung stehen. Dazu gehören nicht nur die Makros selbst, sondern auch spezielle Hilfsmittel zur Datendefinition, die wichtig bei der Arbeit mit dem Assembler sind.

Kapitel 8, 9 und 10 behandeln schließlich noch die Hardware und den „Mikrocode" des IBM PC. Der besondere Nachdruck liegt in diesem Abschnitt auf speziellen Eigenschaften des IBM PC und der Verwendung der Assemblersprache zu ihrer Nutzung. Von besonderem Interesse ist dabei Kapitel 10, in dem die Technik des Zusammenbindens von Assemblerprogrammen mit Programmen in anderen Sprachen oder Systemteilen beschrieben wird. Dabei werden verschiedene Wege der Programmverknüpfung erläutert, ebenso wie die Möglichkeit, Programme direkt in das Betriebssystem einzubinden.

Der Autor des vorliegenden Buches war Mitglied des Entwicklungsteams für den IBM PC. Sein Dank gilt an dieser Stelle all denen in der PC-Abteilung, die bei der Erstellung dieses Buches mit Rat und Tat zur Seite standen. Zu ganz besonderem Dank fühlt er sich seinen Managern Dave O'Connor und Jud McCarthy verpflichtet, sowie vor allem seiner Frau Cynthia für ihre Hilfe und Ermutigung.

David J. Bradley

Inhaltsverzeichnis

4 Der 8088-Befehlssatz

1 Einleitung

Lieber Leser! Dieses Buch soll Sie lehren, Programme in Assemblersprache für den IBM-Personalcomputer zu erstellen. Als Lehrmittel dazu werden wir den Rechner selbst einsetzen.

Programmieren in Assemblersprache

Warum sollten Sie an der Programmierung in Assemblersprache interessiert sein? Moderne, hoch entwickelte Programmiersprachen wie BASIC, FORTRAN, oder PASCAL sind heute weit verbreitet. Wahrscheinlich sind Sie sogar mit einer dieser Programmiersprachen vertraut; und sollten Sie gerade einen IBM PC benutzen, dann wissen Sie, daß der BASIC-Interpreter ein integrierter Bestandteil der Hardware dieses Geräts ist. Warum sich dann mit einer weiteren Programmiersprache befassen, und noch dazu mit einer, die stellenweise erhebliche Schwierigkeiten aufweisen kann? Ganz einfach — auch mit unseren heutigen, hoch entwickelten Programmiersprachen benötigen wir die Assemblersprache wegen ihrer Leistungsfähigkeit und Präzision.

Assemblerprogramme können extrem leistungsfähig sein. Unter gleichen Bedingungen wird ein Assemblerprogramm immer kleiner im Umfang und schneller in der Ausführung sein als das entsprechende Programm in einer höheren Programmiersprache. Dies trifft allerdings nur für kleine und mittelgroße Programme zu. Unglücklicherweise verlieren Assemblerprogramme nämlich mit zunehmender Größe an Leistungsfähigkeit. Dies ist darauf zurückzuführen, daß in Assemblerprogrammen auch die kleinsten Dinge vom Programmierer selbst erledigt werden müssen. Wie Sie im Weiteren sehen werden, erfordert es die Assemblerprogrammierung, jeden einzelnen Schritt des Rechners festzulegen. Dies wird bei kleineren Programmen zu einer optimalen Ausnutzung der Computerhardware führen. Bei größeren Programmen kann Sie jedoch die riesige Menge von Einzelheiten davon abhalten, das Programm optimal zu gestalten, auch wenn Ihnen einige Teile sicherlich sehr gut gelingen werden. Die Assemblerprogrammierung ist also keine absolute Lösung für jede Art von Programm.

Assemblerprogramme sind also sehr stark detailliert. Da es diese Sprache dem Programmierer erlaubt, direkt auf die Hardware zuzugreifen, sind mit Assemblerprogrammen außerdem Dinge möglich, die mit Programmen in höheren Programmiersprachen völlig undenkbar sind. So ist z.B. für die Programmierung von Ein-Ausgabeeinheiten, wo ein Programm einzelne Bits manipulieren muß, Assemblersprache die einzige mögliche Wahl.

Natürlich bieten die Genauigkeit und Leistungsfähigkeit der Assemblersprache Vorteile. Ihr Zwang zum Detail kann aber auch Probleme verursachen. Wann ist es also an der Zeit, die Assemblersprache einzusetzen?

Ganz sicherlich müssen wir die Assemblersprache dort einsetzen, wo es keine anderen Möglichkeiten gibt, Programme zu schreiben. So schrieben z.B. IBM-Programmierer alle Routinen zur Steuerung der peripheren Geräte des IBM PC in

Assembler. Um periphere Geräte und das Interruptsystem zu steuern, war es nämlich nicht möglich, eine andere Sprache einzusetzen. Ebenso wurden von IBM die Diagnoseroutinen, die auch die kleinsten Einzelheiten der Hardware überprüfen müssen, in Assembler geschrieben.

Sie sollten Assemblersprache immer dann verwenden, wenn Ihr Programm besonders leistungsfähig sein muß. Diese Leistungsfähigkeit kann dabei entweder in der Ausführungsgeschwindigkeit Ihres Programms oder aber in seiner Größe liegen. Die mathematischen Subroutinen für FORTRAN sind ein Beispiel für Programme von hoher Leistungsfähigkeit, sowohl was Ablaufgeschwindigkeit als auch Platzbedarf betrifft. Diese mathematischen Routinen sind Teil jedes FORTRAN-Programms, weshalb sie so klein wie möglich sein müssen. Sie bearbeiten aber auch sämtliche mathematischen Funktionen in einem FORTRAN Programm, werden also sehr häufig benutzt, weshalb sie sehr schnell ablaufen sollen.

Welche Programme sollte man nun nicht in Assembler schreiben? Nun, jedes Programm kann in Assemblersprache geschrieben werden, doch von einer bestimmten Programmgröße an ist es besser, eine höhere Programmiersprache zu benutzen, so wie BASIC oder PASCAL. Diese Sprachen ermöglichen es Ihnen, sich nur mit der Lösung Ihres Problems zu beschäftigen. Mit den Einzelheiten der Hardware und des Rechners haben Sie dann nichts mehr zu tun. Eine höhere Programmiersprache erlaubt es Ihnen also, bildlich gesagt, sich mit dem Wald, und nicht mit den Bäumen zu beschäftigen.

Offensichtlich ist es also nötig, Assemblerprogramme mit solchen in höheren Programmiersprachen zu kombinieren. Dieses Buch wird sich besonders mit Assemblerprogrammierung in dafür geeigneten Aufgabengebieten befassen, wie z.B. die Steuerung von peripheren Geräten. Zum Schluß werden wir noch darauf eingehen, wie man Assemblerprogramme mit anderen, in einer höheren Sprache geschriebenen Programmen zusammenbindet. Diese Methode stellt einen optimalen Kompromiß dar. Sie können Assemblerroutinen dort verwenden, wo sie wegen ihrer Leistungsfähigkeit nötig sind, und Routinen in höheren Programmiersprachen für den Rest Ihres Programms, Sie müssen sie nur richtig zusammenbinden.

Es gibt noch einen letzten Grund, das Programmieren in Assemblersprache zu erlernen. Nur durch das Schreiben von Programmen auf dieser untersten Ebene lernen Sie, wie der Rechner in seinem Innersten arbeitet. Wenn Sie wirklich alles wissen wollen, was es über einen Rechner zu wissen gibt, dann müssen Sie mit seiner Assemblersprache vertraut sein. Und der einzige Weg dies zu tun, ist, Programme in dieser Sprache zu schreiben, das reine Lesen dieses Buches wird dazu nicht ausreichen.

Der IBM-Personalcomputer

Warum verwenden wir in diesem Buch den IBM PC als Grundlage zum Erlernen der Assemblersprache? Dafür gibt es mehrere Gründe. Erstens, der IBM PC ist neu und sehr leistungsfähig. Als Personalcomputer hat die IBM-Maschine Fähigkeiten, die

weit über die früherer Rechner hinausreichen. Wie wir später genauer sehen werden, benützt der IBM PC den Intel 8088 Mikroprozessor. Dieser Prozessor ermöglicht 16-Bit-Arithmetik und kann einen Speicherbereich von 1 Million Zeichen direkt adressieren. Diese Eigenschaften verleihen ihm Fähigkeiten, die eher an einen Großrechner also an einen früheren PC erinnern.

Zweitens verfügt der IBM PC über all die Programmierwerkzeuge, die wir zur Erstellung von Assemblerprogrammen benötigen. Neben dem Assembler, um sie einmal aufzuzählen, verfügt der IBM PC über einen Editor, einen Linker, und über das Disk Operating System. Es gibt sogar einen Debugger, d.h. ein Programm zur Fehlersuche, mit dem Sie Ihr eigenes Programm im Fehlerfall bis in die kleinsten Einzelheiten verfolgen können.

Schließlich ist der IBM PC auch noch wegen seiner leichten Erhältlichkeit ein geeignetes System zum Erlernen der Assemblersprache. Er ist eine verhältnismäßig preisgünstige Maschine und verfügt dennoch über alle Eigenschaften, die wir zur Programmierung in Assemblersprache benötigen. Mehr noch, er ist Ihr „persönlicher" Computer, eine Maschine, die Ihnen gehört, zumindest solange sie Ihr Programm ausführt. Dies bedeutet, daß Sie auf ihr Dinge ausprobieren können, die Sie auf einer großen Maschine nie ausprobieren könnten, da Sie diese ja mit anderen Benutzern teilen müssen. Sie können z.B. die Behandlung der Ein-/Ausgabegeräte übernehmen und sie interessante Dinge tun lassen. Sie können mit jedem Teil des Systems tun, was immer Sie wollen. Und Sie können dies sogar tun, selbst wenn das System zusammenbricht. Es ist ja Ihre persönliche Maschine, und wenn es ein Problem gibt, dann schalten Sie die Maschine einfach aus und starten sie wieder. Der einzige mit dem Sie in Schwierigkeiten kommen können, sind Sie selbst. Mit Ihrer persönlichen Maschine haben Sie so eine großartige Spielwiese und zugleich einen hervorragenden Arbeitsplatz zur Entwicklung Ihrer Programme.

Dieses Buch

Dieses Buch führt Sie in Technik und Assemblersprache des IBM PC ein. Obwohl der Schwerpunkt auf der Programmierung in Assemblersprache liegt, werden auch Aspekte der Programmierung wichtiger Teile der Hardware Ihres Rechners behandelt. Sie werden dabei entdecken, wie die Ein-/Ausgabegeräte funktionieren und wie Programme sie korrekt arbeiten lassen. Und Sie werden auch lernen, wie Sie auf dem IBM PC Ihre eigenen Assemblerprogramme schreiben. Zudem zeigt Ihnen dieses Buch, wie Sie Ihre Assemblerroutinen in Programme aus höheren Programmiersprachen einbinden, oder wie Sie sie in das Betriebssystem integrieren.

Assemblerprogrammierung ist ein faszinierendes Erlebnis, kann aber auch frustrierend sein. Einige Programmbeispiele in diesem Buch sind voll funktionsfähig. Diese Beispiele sollen Ihnen ein Anreiz sein. Doch der einzige Weg, Programmierung zu erlernen, ist, selbst Programme zu schreiben. Sie müssen Ihre eigenen Fehler machen, und daraus lernen. Viel Glück und viel Spaß dabei!

2 Computergrundlagen

Dieses Kapitel sollen Ihnen die Eigenschaften eines Computers erklären. Sie erfahren dabei, wie ein Computer arbeitet, und warum er so und nicht anders arbeitet. Einige dieser Prinzipien werden Ihnen sicherlich schon bekannt sein. Falls Sie jedoch noch keine Erfahrung in Assemblerprogrammierung haben, dürften Ihnen viele dieser Prinzipien neu sein.

Binärarithmetik

Alle Rechner speichern Informationen binär. Dies bedeutet, daß jede vom Computer gespeicherte Informationseinheit nur aus zwei Zuständen bestehen kann. Diese Zustände werden als „Ein" und „Aus", „Wahr" und „Falsch" bzw. 1 und 0 bezeichnet. Der Computer speichert diese Zustände in Form von Spannungswerten. Wenn wir Programme schreiben, brauchen wir uns glücklicherweise nur mit den Zahlen, und nicht mit diesen Spannungswerten zu befassen. Bereits mit den einfachen Zahlen 0 und 1 können wir übrigens ganz komplizierte Arithmetik bewerkstelligen.

Wegen der binären Datendarstellung benutzt ein Computer auch für alle seine Rechenoperationen Binärarithmetik. Binärarithmetik arbeitet nur mit zwei Ziffern, nämlich 0 und 1. Im normalen Umgangsleben benutzen wir Zehner- oder Dezimalarithmetik. In der Dezimalarithmetik stehen uns zehn verschiedene Ziffern, nämlich die Ziffern 0–9, zur Verfügung. Binärarithmetik dagegen kann man sich als Rechensystem für Leute vorstellen, die nur zwei anstelle von zehn Fingern besitzen.

Die Beschränkung auf zehn Ziffern im Dezimalsystem hindert uns nicht daran, damit wesentlich größere Zahlen darzustellen. Wir benutzen in diesem Fall mehrstellige Zahlen, wobei jede Ziffernstelle einer solchen Zahl eine andere Zehnerpotenz darstellt. Die niedrigste oder auch äußerst rechte Stelle einer solchen Zahl ist die Einerstelle. Eins weiter links finden wir die Zehnerstelle, die nächste Stelle ist die Hunderterstelle, usw.. Die Steigerung von rechts nach links erfolgt also in Zehnerpotenzen. Die Zahl 2368 besteht in der Tat aus 2 Tausendern, 3 Hundertern, 6 Zehnern und 8 Einern. Abbildung 2.1 zeigt Ihnen die Zerlegung der Zahl 2368 auf mathematische Art.

$$2368 = 2 \times 10^3 + 3 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$$
$$= 2000 + 300 + 60 + 8$$

Abbildung 2.1 Dezimale Zahlendarstellung

Binärarithmetik funktioniert genauso, nur daß die Ziffernstellen einer Zahl hier Zweierpotenzen und nicht Zehnerpotenzen darstellen. Alle Zahlen größer als 1 werden dabei mehrstellige Zahlen, ebenso wie im Zehnersystem jede Zahl größer als 9 eine mehrstellige Zahl wird. Jede Stelle einer Binärzahl wird als Bit bezeichnet, stellvertretend für „binary digit". Die Position jedes Bits innerhalb einer Binärzahl entspricht seiner Zweierpotenz. Abbildung 2.2 zeigt die Aufschlüsselung der Binärzahl 101101B.

$$
\begin{aligned}
101101B &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 32 \quad + 0 \quad + 8 \quad + 4 \quad + 0 \quad + 1 \\
&= 45
\end{aligned}
$$

Abbildung 2.2 Binäre Zahlendarstellung

Wir werden den Buchstaben B in Zukunft benützen, um anzuzeigen, daß eine Zahl eine Binärzahl ist. So können wir Dezimalzahlen, die nicht besonders gekennzeichnet sind, von Binärzahlen unterscheiden. Die Zahl 2368 ist beispielsweise eine Dezimalzahl, während 101101B eine Binärzahl ist. Im Gegensatz zu unserer Schreibweise benützen die meisten mathematischen Fachbücher tiefgestellte Ziffern zur Kennzeichnung des jeweils verwendeten Zahlensystems. Wir jedoch verwenden den Buchstaben B zur Kennzeichnung von Binärzahlen, da auch der IBM Assembler dieses Zeichen verwendet.

Die Tabelle in Abbildung 2.3 zeigt, daß 4 Bit Binärdaten benötigt werden, um die dort aufgeführte größte Dezimalzahl darzustellen. Bei größeren Zahlen werden entsprechend mehr Bits benötigt. Eine Binärzahl mit n Bits kann eine Dezimalzahl in der Größenordnung von $2^n - 1$ darstellen. Das bedeutet, daß eine Binärzahl in der Länge von n Bits nur Dezimalzahlen in der Größenordnung von 0 bis $2^n - 1$ darstellen kann. Für das 4-Bit-Beispiel in Abbildung 2.3 ist 15 ($2^4 - 1$) die größte darstellbare Dezimalzahl.

Dezimal	**Binär**
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Abbildung 2.3 Die ersten zehn ganzen Zahlen

Für jeden Typ von Mikroprozessor gibt es eine bestimmte maximale Größe von Binärzahlen, die er verarbeiten kann. Für den 8088, der im IBM PC verwendet wird, beträgt diese interne Datenbreite 16 Bits. Die größte Dezimalzahl, die mit 16 Bits dargestellt werden kann, ist $2^{16} - 1$, oder 65.535. Diese Art, mit vorzeichenlosen ganzen Zahlen zu arbeiten, ermöglicht uns die Darstellung von Zahlen zwischen 0 und 65.535. Wir benötigen nun noch eine Abwandlung dieses Systems, um auch negative Zahlen darstellen zu können.

Zweierkomplement

Um sowohl positive als auch negative Zahlen darstellen zu können, benutzt der 8088 Arithmetik im Zweierkomplement. In dieser Rechenart stellt das ganz links stehende Bit einer Zahl das Vorzeichen dar. Positive Zahlen haben eine 0 an ihrer höchstwertigen Stelle, während negative Zahlen dort eine 1 haben. Positive Zahlen werden sowohl mit als auch ohne Vorzeichen gleich dargestellt. Bei negativen Zahlen sieht die Sache etwas anders aus. Um eine Zahl zu negieren, d.h. ihr Vorzeichen zu verändern, wird die Zahl komplementiert und das Ergebnis um 1 erhöht. In einer 4-Bit-Darstellung hat die Zahl 5 z.B. den Wert 0101B, während die Zahl −5 den Wert 1011B besitzt. Das Beispiel in Abbildung 2.4 erklärt dise Vorgehensweise.

In Abbildung 2.5 sehen wir, daß auch in der Darstellung über das Zweierkomplement die Null nur einmal auftaucht. Das bedeutet, −0 entspricht +0. Für eine beliebige, n Bit große Zahl im Zweierkomplementsystem ist der größte darstellbare Wert $2^{n-1}-1$, und der kleinste darstellbare Wert -2^{n-1}. Die Null taucht also nur einmal auf. Dadurch ist z.B. in einem 4-Bit-System die größte darstellbare Zahl 7, während die kleinste −8 ist, wie in Abbildung 2.6 gezeigt.

5 =	0101B
Zum Vorzeichenwechsel:	
Komplementieren	1010B
Addieren von 1	0001B
−5 =	1011B
Zum Vorzeichenwechsel:	
Komplementieren	0100B
Addieren von 1	0001B
5 =	0101B

Abbildung 2.4 Zweierkomplement von 5

0 =	0000B	
Zum Vorzeichenwechsel:		
Komplementieren	1111B	
Addieren von 1	0001B	
−0 =	0000B	= 0

Abbildung 2.5 Nulldarstellung

Wie wir später sehen werden, kann der 8088 ganze Zahlen sowohl vorzeichenbehaftet wie auch ohne Vorzeichen verarbeiten. Die Wahl bleibt dem Assemblerprogrammierer überlassen. Findet in einem Programm jedoch vorzeichenbehaftete Arithmetik Verwendung, so arbeitet der 8088 mit der Zeichendarstellung im Zweierkomplement.

Dezimal	Binär	Dezimal	Binär
7	0111	−1	1111
6	0110	−2	1110
5	0101	−3	1101
4	0100	−4	1100
3	0011	−5	1011
2	0010	−6	1010
1	0001	−7	1001
0	0000	−8	1000

Abbildung 2.6 Zahlen im Zweierkomplement

Hexadezimale Zahlendarstellung

Binärarithmetik ist hervorragend geeignet für Computer, denn sie arbeiten nur mit Einsen und Nullen. Für uns Menschen benötigen wir allerdings eine etwas gedrängtere Darstellungsart. Wir haben deshalb zu unserer Erleichterung die hexadezimale Zahlendarstellung ausgewählt.

Hexadezimale Zahlendarstellung beruht auf der Abbildung von Zahlen in einem System mit der Basis 16. Jede Ziffernstelle einer Zahl kann also die Werte von 0 bis 15 annehmen. Eine Ziffernstelle entspricht somit jeweils einer Potenz von 16. Diese Hexadezimaldarstellung ist hervorragend zur Wiedergabe binärer Informationen geeignet. Eine Hexadezimalziffer entspricht nämlich genau 4 Bits. Um Zahlen vom Binär- ins Hexadezimalsystem umzuwandeln, müssen wir nur die einzelnen Bits in Gruppen zu jeweils vier Bits aufteilen und dann jede dieser neuen Gruppen als hexadezimale Ziffer lesen. Dies ergibt eine für uns sehr angenehme Verkürzung von 4:1 in der Zahlendarstellung.

Hexadezimalarithmetik stellt allerdings ein kleines Problem dar, denn die einzigen einstelligen Zahlen, die uns zur Verfügung stehen, sind die Zahlen 0 bis 9. Die Zahlen 10 bis 15 stellen wir deshalb durch die ersten sechs Buchstaben des Alphabets, die Zeichen A bis F, dar. Die Entsprechung zwischen dezimalem, hexadezimalem und binärem Zahlensystem ist in Abbildung 2.7 dargestellt.

Dezimal	Binär	Hexadezimal	Dezimal	Binär	Hexadezimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Abbildung 2.7 Hexadezimalzahlen

Wie die Abbildung zeigt, entspricht dabei jede hexadezimale Ziffer genau 4 Bits einer Binärzahl. Hexadezimale Zahlendarstellung ist also immer dann besonders angebracht, wenn die Wortgröße des verwendeten Rechners ein Vielfaches von 4 ist. Da der Intel 8088 eine Wortgröße von 16 Bits aufweist, werden also auch wir die hexadezimale Zahlendarstellung verwenden. Jeder 16-Bit-Wert wird dabei durch 4 hexadezimale Ziffernstellen abgebildet. Im weiteren werden wir jeden Wert in hexadezimaler Darstellung mit dem Suffix „H", und jeden Binärwert mit dem Suffix „B" kennzeichnen. Dezimalzahlen werden je nach Notwendigkeit mit oder ohne Suffix „D" dargestellt. Diese Darstellungsart entspricht übrigens genau der, die wir auch beim Umgang mit Zahlen im Assembler benötigen. In einem Programm können wir jedes der drei besprochenen Systeme (dezimal, binär, und hexadezimal) verwenden, um bestimmte Werte darzustellen.

Beim Schreiben von hexadezimalen Zahlen ist es wichtig, daß der Assembler sie auch als solche auffaßt. Geben wir nämlich den Wert „FAH" ein, so könnte dies entweder die hexadezimale Zahl „FA" oder aber auch die Variable „FAH" sein. Der Assembler nimmt nun immer an, daß eine Zahl mit einer Ziffer, und eine Variable mit einem Buchstaben beginnt. „FAH" ist also für den Assembler der Name einer Variablen. Wollen wir nun, daß der Assembler diesen Wert als Hexadezimalzahl versteht, so müssen wir „0FAH" eingeben, was genauso dem gewünschten Wert entspricht, aber mit einer Ziffer beginnt. Wir müssen also allen hexadezimalen Werten, die mit den Ziffern A bis F beginnen, jeweils eine Null voranstellen, um zu verhindern, daß sie vom Assembler als die Namen von Variablen interpretiert werden.

Maschinensprache und Assembler

Wir haben bisher gesehen wie eine Folge von Nullen und Einsen für den Computer ein Zahlensystem darstellt. Im weiteren werden wir sehen, wie eine ebensolche Folge von Nullen und Einsen dazu verwendet werden kann, einen Computer zu programmieren.

Ein Computerprogramm ist eine Folge von Anweisungen. Diese Anweisungen erklären dem Computer, welche Arbeit er zu verrichten hat. Man könnte dies auch mit einem Rezept in einem Kochbuch vergleichen. Ein Rezept besteht aus einer fortlaufenden Folge von Anweisungen, die erklären, was zu tun ist, um ein bestimmtes Gericht herzustellen. In ähnlicher Weise hat auch der Computer eine bestimmte Folge von Anweisungen, die ihm genau erklären, was er zu tun hat. Diese Folge von Anweisungen nennen wir Programm. Die Kunst, eine korrekte Folge von Anweisungen für einen Computer zu erstellen, wird gemeinhin als Programmieren bezeichnet. In unserem Beispiel mit dem Rezept entspricht das Rezept dem Programm, und die Person, die das Rezept schrieb, dem Programmierer. Der Computer übernimmt dann die Rolle der Köchin.

Das vom Computer tatsächlich auszuführende Programm besteht im Grunde nur aus einer Folge von Nullen und Einsen im Speicher des Rechners. Diese Bitfolge wird gemeinhin als Maschinensprache bezeichnet. Maschinensprache ist also

genau die Sprache, die die Maschine wirklich versteht. Der Computer liest die Anweisungen in Maschinensprache in einer genau definierten Art aus dem Speicher. Danach führt er die durch das jeweilige Bitmuster bestimmten Funktionen aus. Auf diesen Lese-Ausführungs-Zyklus werden wir später genauer eingehen.

Im allgemeinen ist die Verwendung der reinen Maschinensprache wenig sinnvoll. Falls Sie z.B. im 8088 zwei Zahlen addieren wollen (z.B. das AX-Register auf das BX-Register — eine kurze Erklärung der Register wird gleich folgen), würde dies wie folgt aussehen:

0000001111000011B (oder 03C3H).

Diese beiden Bytes sagen dem Computer ganz genau, was er hier zu tun hat. Ähnlich würde die Subtraktion zweier Zahlen (Register BX wird vom AX-Register abgezogen) aussehen:

0010101111000011B (oder 2BC3H).

Ein kurzer Hinweis auf die Register ist hier angebracht, da sie allgemein sehr häufig erwähnt bzw. verwendet werden. Ein Register ist ein ganz spezieller Speicherplatz innerhalb des Prozessors. Der Prozessor kann Daten in Registern sehr schnell erreichen — sehr viel schneller als Daten im normalen Speicher. In Verbindung mit bestimmten Maschinenanweisungen haben Register auch noch besondere zusätzliche Aufgaben. Im Kapitel 3 werden wir die Register des 8088 genau erklären.

Obwohl die Maschinensprache wirklich das Größte für einen Rechner ist, ist sie doch verhältnismäßig schwierig für einen Menschen. Glücklicherweise gibt es aber eine einfachere Methode, einen Rechner (fast) in Maschinensprache zu programmieren. Diese Methode, mehr auf den Menschen zugeschnitten als auf die Maschine, ist das Programmieren in Assembler.

Der Assembler ist eine Programmiersprache, die für den Programmierer wesentlich aussagefähiger ist als die Maschinensprache, obwohl sie direkt der Maschinensprache entspricht. Der Computer liest dazu das Assemblerprogramm und übersetzt es in Maschinensprache, eine Form, die er versteht. Dieser Vorgang, Assemblieren genannt, ist tatsächlich eine Form von Sprachübersetzung. Ein als „Assembler" bezeichnetes Maschinenprogramm übernimmt dabei die Aufgabe, das Assemblerprogramm in Maschinensprache zu übersetzen.

Um den Unterschied deutlich zu machen, sehen wir uns nocheinmal das vorher gezeigte Beispiel an. Die Assembleranweisung, um die Register AX und BX zu addieren, lautet ganz einfach:

ADD AX,BX

Entsprechend, um das Register BX von AX zu subtrahieren, schreiben wir:

SUB AX,BX

Der Assembler übersetzt diese Anweisungen in die Form, die wir bereits vorhin gesehen haben. Der Computer bewältigt hier also das Problem, einen menschenlesbaren Text in ein Maschinenprogramm zu übersetzen, das der Prozessor später ausführen kann.

Assembler ist keine Programmiersprache wie FORTRAN, COBOL, oder Pascal. Diese Sprachen und viele andere auch, sind höhere Programmiersprachen. Diese höheren Programmiersprachen sind genau auf die Probleme zugeschnitten, die mit ihnen gelöst werden sollen. Als solche werden sie manchmal auch prozedurale Sprachen genannt, da sie Prozeduren beschreiben, die dazu dienen, das gegebene Problem zu lösen. Höhere Programmiersprachen sind im allgemeinen maschinenunabhängig. Ein Programm, das in FORTRAN für den IBM PC geschrieben ist, sollte auch dann korrekt laufen und die gleichen Resultate erzeugen, wenn es auf einem IBM-System/370 abläuft. Die Programmiersprache ist also unabhängig vom Rechner.

Assemblerprogramme dagegen sind direkt an die Maschine gebunden, auf der sie ausgeführt werden sollen. Die Assemblersprache ist also maschinenabhängig. Der Assembler für den IBM PC z.B. ist völlig anders als die Assemblersprache für das System/370. Dies rührt daher, daß Assembleranweisungen im allgemeinen direkt in Maschinenanweisungen umgewandelt werden. Das bedeutet auch, daß eine Assembleranweisung normalerweise genau eine Maschinenanweisung ergibt. Nachdem nun die Maschinensprache der einzelnen Computer verschieden ist, sind auch die entsprechenden Assemblersprachen unterschiedlich.

Im allgemeinen erzeugt jede Assembleranweisung einen Maschinenbefehl. Es gibt allerdings auch einige Ausnahmen. Dies rührt daher, daß es Assembleranweisungen gibt, die nur für den Assembler bestimmt sind, und die nicht Teil des späteren Maschinenprogramms werden sollen. Diese Anweisungen sind also direkte Steueranweisungen für den Assembler. Sie werden zur Zeit der Assemblierung ausgewertet. Ein Beispiel für eine solche Assembleranweisung ist

TITLE Beispielprogramm

Die Anweisung teilt dem Assembler nur den Namen des zu übersetzenden Programms mit. Nach der Übersetzung des Programms durch den Assembler wird dieser Titel — Beispielprogramm — zu Beginn jeder Seite der vom Assembler erzeugten Liste stehen. Die Anweisung hat also nur für den Assembler Bedeutung. Es gibt keinen Befehl für den 8088, der eine ähnliche Leistung erbringen könnte.

Syntax der Assemblersprache

Bevor wir weitergehen, müssen wir zunächst einmal die Syntax der Assemblerbefehle besprechen. Wir müssen die einzelnen Teile der Assemblersprache genau und verbindlich definieren, so daß uns für die Zukunft ein eindeutiger Satz von Bezeichnungen für diese Teile der Sprache zur Verfügung steht.

PART 1:	ADD	AX,BX	; Addieren Pufferlänge
Label	Opcode	Operanden	Kommentar

Abbildung 2.8 Syntax der Assemblersprache

Ein Assemblerbefehl besteht aus bis zu vier Teilen. Abbildung 2.8 zeigt einen typischen Assemblerbefehl mit der Bezeichnung seiner Einzelteile.

Der einzige notwendige Teil eines Assemblerbefehls ist der Opcode, was eine Zusammensetzung aus „Operation Code" darstellt, zu deutsch auch: Befehlscode. Von manchen Leuten wird auch der Befehlssatz eines Rechners solchermaßen als Satz von Opcodes bezeichnet. Der Opcode-Teil eines Assemblerbefehls teilt dem Prozessor mit, was er zu tun hat, im Beispielsfall also, eine Additionsanweisung (ADD) auszuführen.

Das Operandenfeld enthält zusätzliche Informationen über den auszuführenden Befehl — z.B. welche Werte an der Operation beteiligt sind. Das Operandenfeld wird dabei durch den Opcode bestimmt. Jeder Opcode benötigt einen bestimmten Satz von Operanden. Eine Additionsanweisung (ADD) benötigt z.B. zwei Operanden, — die beiden Werte, die addiert werden sollen. Eine Negationsanweisung (NEG) dagegen benötigt z.B. nur einen Operanden, und einige Anweisungen, wie z.B. der Dezimalangleichungsbefehl DAA, benötigen überhaupt keinen Operanden. In Kapitel 4 werden wir alle diese Operationen und Operanden besprechen.

Die Label- und Kommentarfelder sind bei jeder Anweisung optional. Das Labelfeld stellt eine Möglichkeit dar, einer bestimmten Speicherstelle innerhalb des Computers einen Namen zu geben. Jede Speicherstelle in einem Rechner hat eine eindeutige Adresse, doch die Adresse eines Befehls festzustellen, ist für den Benutzer schwierig, wenn nicht gar unmöglich. Ein Label erlaubt es nun, über einen frei vergebenen Namen eine bestimmte Adresse im Speicher zu identifizieren. Wir könnten auch sagen, das Labelfeld enthält die symbolische Adresse des damit bezeichneten Befehls. Wollen wir später wieder auf diesen Punkt im Befehlsablauf zugreifen, so tun wir das über diesen symbolischen Namen. Die tatsächliche Adresse des Befehls im Speicher benötigen wir dann nicht. Die Möglichkeit solcher Labels ist einer der Gründe, warum wir die Assemblersprache der Maschinensprache vorziehen. Der Assembler bewerkstelligt nämlich für uns die Umwandlung symbolischer Namen in tatsächliche Maschinenadressen.

Das Kommentarfeld ist für den Programmierer gedacht. Er kann in diesem Bereich zusätzliche Angaben über den gegebenen Befehl machen. Kommentare müssen nicht auf einen Befehl beschränkt bleiben. Wir können ganze Kommentarzeilen in ein Programm einfügen, indem wir an die erste Stelle einer solchen Zeile ein Semikolon setzen. Auf diese Weise können wir ganze Informationsblöcke in unser Programm einfügen, z.B., um verwendete Algorithmen darzustellen.

Jeder hat seine eigenen Ideen, wie ein Programm kommentiert werden sollte, und auch Sie werden sicherlich Ihre eigenen Vorstellungen davon haben. Im allgemeinen sollte man Informationen einfügen, die sich auf das zu lösende Problem beziehen. In unserem gezeigten Beispiel wäre es z.B. sinnlos, den Befehl mit einem Kommentar zu versehen wie „Addiere AX auf BX". Das wäre nichts anderes als eine Wiederholung dessen, was durch Opcode und Operanden bereits dargestellt ist. Wenn Sie sich schon mit einem Kommentar belasten, dann sollte er auch die Mühe wert sein, ihn zu lesen bzw. zu schreiben.

Arbeitsweise des Assemblers

Sehen wir uns nun an, was der Assembler ganz allgemein tut. Die Feinheiten werden wir später entdecken, doch zunächst brauchen wir noch einige neue Begriffe und sollten uns auch einmal einen Output ansehen.

Der Assembler übersetzt ein in Assemblersprache geschriebenes Programm in Maschinensprache. Die Eingabedatei, die das Programm in Assemblersprache enthält, wird dabei als Quelldatei bezeichnet. Der Output des Assemblers ist noch keine reine Maschinensprache, sondern nur eine Zwischenform. Diese Ausgabedatei wird als Objektdatei bezeichnet. Die Daten, die sie enthält, sind der Objektcode. Dieser Objektcode muß erst noch modifiziert werden, um echte Maschinensprache darzustellen. Im Fall des IBM PC bewerkstelligt dies das LINK-Programm. Die Erweiterung des Objektcodes zur vollständigen Maschinensprache wird gemeinhin als Linken bezeichnet. Die Benutzung des LINK-Programms werden wir in einem späteren Kapitel erklären.

Neben der Umwandlung des Quellcodes in Objektcode erzeugt der Assembler auch noch eine zusätzliche Ausgabedatei. Eine davon ist die List-Datei. Sie umfaßt alle Aktionen des Assemblers und enthält den originalen Quellcode zusammen mit allen Kommentaren. Sie enthält außerdem den vom Assembler erzeugten Objektcode. Abbildung 2.9 ist ein Beispiel für eine Assemblerliste, die manchmal auch als Print-Datei bezeichnet wird.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 2.9  Assembly Example

1                                         PAGE    ,132
2                                         TITLE   Figure 2.9  Assembly Example
3       0000                    CODE      SEGMENT
4                                         ASSUME  CS:CODE
5
6       0000  03 C3             PART1:    ADD     AX,BX          ; Add in the buffer length
7
8       0002                    CODE      ENDS
9                                         END
```

Abbildung 2.9 Assemblerprogramm

Nehmen wir nun einmal unseren Beispielbefehl, und sehen wir uns an, was der Assembler daraus macht. Auf der rechten Seite der Liste sehen wir unseren Originalbefehl. Auf der linken Seite stehen die Informationen, die der Assembler daraus generiert. Die erste Spalte enthält die fortlaufende Zeilennummer jeder Zeile innerhalb der Liste. Der Assembler versieht jede Zeile der Quelldatei mit einer solchen Nummer. Diese Zeilennummern müssen dabei nicht notwendigerweise mit den Nummern übereinstimmen, die vielleicht irgendein Editor für die Quelldatei vergibt.

Die zweite Spalte stellt die Adresse des Befehls dar. Der Linker kann später die Adresse modifizieren. Doch zunächst einmal stellt sie den besten Vorschlag dar, den der Assembler während der Übersetzung machen kann. Die nächste Spalte

beinhaltet den Maschinencode des Befehls. Da die Befehle des 8088 zwischen 8 und 56 Bits in der Länge schwanken können, kann dieses Feld variable Länge haben. Auch der Linker kann später noch Änderungen an diesem Objektcode vornehmen. Er kann ganz allgemein jeden Teil eines Befehls verändern, der sich auf eine Adresse bezieht. Mit Ausnahme der Adressfelder stellt eine Assemblerliste also ein genaues Ebenbild des Maschinencodes dar, wie er später zur Ausführung kommen wird.

Für die meisten unserer weiteren Beispiele werden wir die Assemblerliste als Grundlage verwenden. Dies wird uns in die Lage versetzen, jederzeit den vom Assembler erzeugten Code einzusehen.

Die zweite vom Assembler erzeugte Datei ist die Querverweisdatei. Diese Datei enthält alle Querverbindungen zwischen Labels und Anweisungen, die sich auf diese Labels beziehen. Diese Information ist von beinahe unschätzbarem Wert, sobald wir versuchen, ein Programm zu ändern. Über die Querverweisliste können wir nämlich all die Anweisungen bestimmen, die eine bestimmte Adresse im Speicher ansprechen. Dies erlaubt es uns, all die Befehle zu bestimmen, die in Mitleidenschaft gezogen werden könnten, sobald wir einen Teil unseres Programmes ändern. In Kapitel 5 werden wir genauer auf die Verwendung dieser Querverweisinformationen eingehen.

Über Bits, Bytes und Wörter

Wir verwenden den Namen „Bit" für „binary digit", das einmalige Auftreten eines der beiden Werte 0 oder 1. Aus Gründen der Einfachheit werden wir nun auch für einige Kombinationen von Bits eigene Namen vergeben.

Eine Einheit von 8 Bits wird gemeinhin als Byte bezeichnet. In allen IBM-Unterlagen und auch in diesem Buch werden wir jede Einheit von 8 Bits als Byte bezeichnen. Das Byte verdient seinen besonderen Namen aus mancherlei Gründen. So umfaßt z.B. eine Speicherzelle 8 Bits. Oder so macht z.B. jeder Speicherzugriff innerhalb des PC dem Prozessor genau 8 Informationsbits verfügbar. Wie wir später noch sehen werden, können bestimmte Befehle des 8088 arithmetische oder logische Operationen auf 8-Bit-Einheiten ausführen. Ein Byte ist außerdem die kleinste Dateneinheit, die der 8088 direkt bearbeiten kann. So kann er zwei 8-Bit-Zahlen in einer einzigen Anweisung addieren, aber er kann dies nicht mit zwei 4-Bit-Zahlen. Der IBM PC benützt Bytes zusätzlich auch für die Datendarstellung von Zeichen. Ein Byte kann 256 (2^8) eindeutige, aber verschiedene Werte annehmen, wie z.B. Buchstaben. Im nächsten Abschnitt werden wir uns einmal den Zeichensatz des IBM PC ansehen.

Da ein Byte eine Speicherzelle darstellt, sollten wir auch über eine Möglichkeit verfügen, den Inhalt solcher Speicherzellen zu bestimmen. Und in der Tat besteht eine der Aufgaben des Assemblers auch darin, den Inhalt des Speichers für die Ausführung eines Programms zu bestimmen. Normalerweise besteht ein Assemblerprogramm aus ausführbaren Anweisungen. Doch um einen bestimmten Wert in ein bestimmtes Byte zu plazieren, verfügt der Assembler über einen speziellen Mecha-

nismus, die „Define-Byte" oder DB-Pseudoanweisung. DB ist kein 8088-Befehl. Es ist nur eine Anweisung an den Assembler, einen bestimmten Wert im Speicher abzulegen. Der Pseudobefehl

```
DB  23
```

weist den Assembler nur an, den dezimalen Wert 23 an der Speicherstelle abzulegen, deren Adresse gerade aktuell ist.

```
DB  1,2,3,4
```

speichert z.B. die Werte 1 bis 4 an vier aufeinanderfolgende Speicherstellen.

Assemblerprogramme verwenden die DB-Anweisung also, um Datenbereiche zu definieren. In den bisherigen Beispielen haben wir bestimmte feste Werte im Speicher abgelegt. Dies könnte z.B. eine Tabelle oder eine Umwandlungsinformation sein. Wir werden uns noch weitere Beispiele ansehen, die solchermaßen festgelegte Daten verwenden. Es gibt aber auch Möglichkeiten, wo ein Programm Speicherplatz benötigt, um Daten während der Ausführung abzulegen. Zur Zeit der Assemblierung eines solchen Programms ist der Wert der jeweiligen Speicherstellen unbekannt — ihr Inhalt wird auch während der Ausführung des Programms veränderlich sein. Der Befehl

```
DB  ?
```

weist den Assembler an, ein Byte zu reservieren, doch den Inhalt dieses Bytes nicht zu bestimmen. Ein solchermaßen definiertes Byte wird solange einen beliebigen Wert beinhalten, bis ein Befehl einen bestimmten Wert dort abspeichert.

Wir können solchermaßen auch eine größere Anzahl von Speicherstellen reservieren — z.B. um Speicherplatz für ein Array zu reservieren. Wir bewerkstelligen dies mit der Anweisung

```
DB  25 DUP(?)
```

die 25 Bytes Speicherplatz reserviert. Das Schlüsselwort in diesem Pseudobefehl ist DUP, was soviel wie duplizieren bedeutet. Der Wert 25 bestimmt die Anzahl der Wiederholungen für die vom Assembler auszuführende DB-Operation. Der Assembler benützt außerdem die Werte innerhalb der Klammern, um den Speicherbereich vorzubelegen. In unserem Fall ist der Wert unbekannt. Um z.B. einen Bereich mit einem einzigen Wert zu belegen, erzeugt der Befehl

```
DB  17 DUP(31)
```

eine Reihe von 17 aufeinanderfolgenden Bytes, von denen jedes den Wert 31 enthält. Und zu guter Letzt erzeugt der Befehl

```
DB  30 DUP(1,2,3,4,5)
```

eine Reihe von 30 Bytes mit den Werten 1 — 5 in den ersten 5 Bytes. Die nächsten 5 Bytes enthalten wiederum die Werte 1 — 5, und so fort. Der Assembler wiederholt also die Werte in den Klammern bis schließlich alle 30 Bytes belegt sind.

Manchmal wird die Notwendigkeit auftauchen, eine Bitgruppe, die kleiner ist als ein Byte, anzusprechen. Ein gängiger Wert dafür ist 4 Bits. Wir können z.B. die 10 Dezimalziffern in jeweils 4 Bits darstellen. Für solche Bitgruppen werden wir in Zukunft den Ausdruck „Nibble" verwenden. Dieser Ausdruck, der weit verbreitet ist, erlaubt es uns, auch Einheiten, die kleiner als 1 Byte sind, anzusprechen.

Ein „Wort" hat für einen Programmierer eine gänzlich andere Bedeutung als für einen englischen Major. In Hinsicht auf Computer ist ein Wort die größte Anzahl von Bits, die der Rechner in einem Befehl bearbeiten kann. Beim IBM System/370 ist ein Wort z.B. 32 Bits lang. Beim Intel 8088 enthält ein Wort 16 Bits. Deshalb bleibt ein Wort eine zweideutige Bezugseinheit, solange der zugehörige Rechner nicht bekannt ist.

Die Wortgröße des 8088 beträgt also 16 Bits. Die internen Datenwege des Prozessors bestimmen diese Größe. Der 8088 kann arithmetische und logische Operationen mit Zahlen bis zu einer Größe von 16 Bits in einem einzigen Befehl vollziehen. Jede größere Zahl benötigt mehr als einen Befehl. Es gibt auch Befehle, die kleinere Bitmengen verarbeiten, so z.B. die Addition von zwei 8-Bit-Zahlen. Und es gibt einige wenige Befehle, die es erlauben, einzelne Bits zu manipulieren. Die Addition von zwei 32-Bit-Zahlen benötigt jedoch zwei Befehle, wobei jeweils 16 Bits mit einem Befehl bearbeitet werden. Die größte Zahl, mit der wir allgemeine Operationen wie z.B. eine Addition ausführen können, ist die Zahl, die gerade noch in einem Maschinenwort Platz findet.

So wie es einen Assemblerbefehl zur Definition eines Bytes im Speicher gibt, gibt es auch einen Befehl zur Definition eines Worts. Der Assemblerbefehl hierzu lautet DW, für „Define Word". So bestimmt der erste DW-Befehl in Abbildung 2.10 eine 16-Bit Speichereinheit mit dem Wert 1234H. So wie mit Bytes können wir auch bei Wörtern die DUP-Angabe zur Definition größerer, zusammenhängender Speicherbereiche verwenden. Ebenso verwenden wir die Angabe "?" zum Kennzeichnen nicht vorbelegter Speicherbereiche.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 2.10 Define Word Examples

 1                                              PAGE    ,132
 2                                              TITLE   Figure 2.10 Define Word Examples
 3
 4          0000  1234                          DW      1234H
 5          0002     03 [                       DW      3 DUP(5678H)
 6                          5678
 7                                ]
 8
 9          0008  ????                          DW      ?
10
11                                              END
```

Abbildung 2.10 Beispiel für Define Word (DW)

Eine der verwirrendsten Eigenschaften des 8088 ist seine Methode, Informationen mit Wortlänge im Speicher abzulegen. In Abbildung 2.10 sehen wir, daß der von uns definierte Wert 1234H vom Assembler im Speicher als 3412H abgelegt wird. Wie kommt das?

Nehmen wir an, das Wort mit dem Inhalt 1234H befindet sich an den Speicherstellen 100 und 101. Der 8088 verlangt nun den Wert 34H an Stelle 100 und den Wert 12H an Stelle 101. Die einfachste Erklärung ist, daß der Assembler das niedrigstwertige Byte, also die Hexadezimalziffern 3 und 4, an der niedrigeren Adresse, und das höchstwertige Byte, also die Hexadezimalziffern 1 und 2, an der höheren Adresse ablegt. Abbildung 2.11 zeigt den Speicherinhalt nach Ablegen der Daten durch den Assembler.

Speicherstelle	Wert
⋮	⋮
100	34H
101	12H
⋮	⋮

Abbildung 2.11 Speicherdarstellung von DW 1234H

Solange Sie sich noch nicht gänzlich an diese Eigenheit gewöhnt haben, wird es Ihnen immer so erscheinen, als ob alle Informationen im Speicher verkehrt herum stünden. Glücklicherweise brauchen Sie sich nicht um dieses offensichtliche „Byte tauschen" kümmern, solange Sie nicht mit Byte- und Wortoperationen gleichzeitig auf denselben Speicherbereich zugreifen. Ein Programm kann ohne Einschränkungen mit Wortwerten arbeiten, und der 8088 wird die Befehle auch immer richtig ausführen. Nur wenn Sie ein bestimmtes Byte eines Wortwertes ansprechen wollen, werden Sie sich mit der Methode befassen müssen, mit der der 8088 Wörter im Speicher ablegt. Der Assembler betont die Wortstruktur auch in der Programmliste. So werden Wörter im Objektcode als Wörter und nicht als Bytes ausgegeben, die ja in umgekehrter Reihenfolge erscheinen würden. Wir können ein Wort auch daran erkennen, daß der Assembler es als eine Folge von 4 Hexadezimalzeichen ohne Zwischenraum darstellt.

Es gibt noch einen weiteren Datentyp, der in Assemblerprogrammen für den 8088 häufig verwendet wird. Dies ist der 32-Bit-Datenwert, oder das Doppelwort. In Programmen benützt man Doppelwörter um z.B. Adresswerte oder sehr große Zahlen zu speichern. Zum Definieren eines Datenbereichs für ein Doppelwort dient die Assembleranweisung

DD Wert

Sie erzeugt einen 4-Byte-Bereich. Dabei steht DD für „Define Doubleword". So wie beim DW-Befehl legt auch hier der Assembler das niedrigstwertige Byte an der niedrigsten Adresse, und das höchstwertige Byte an der höchsten Adresse ab. Die beiden mittleren Bytes werden entsprechend angeordnet. Wie beim DB- und DW-Befehl können wir auch hier die DUP-Funktion verwenden und die Angabe „?" machen, um einen Speicherbereich undefiniert zu lassen.

Es gibt noch weitere Datenstrukturen, die der Assembler erzeugen kann. Wir werden aber erst dann auf sie genauer eingehen, wenn wir den Makro-Assembler und den 8087 besprechen. In Programmen finden diese Datenstrukturen hauptsächlich Verwendung für extrem große Zahlen in Verbindung mit dem Arithmetikprozessor oder zur Definition benutzereigener Datenstrukturen.

Die Numerierung der Bits

Es wird vorkommen, daß wir bestimmte einzelne Bits innerhalb eines Bytes oder Worts ansprechen wollen. Um dies zu ermöglichen, numerieren wir die Bits. Der Index oder die Nummer, die jedes Bit erhält, ist eine Potenz von 2, die zugleich auch der Position des Bits innerhalb des Bytes oder Worts entspricht. Das niedrigstwertige Bit ist also Bit 0, da es 2^0 darstellt. Das höchstwertige Bit in einem Byte ist Bit 7, also 2^7.

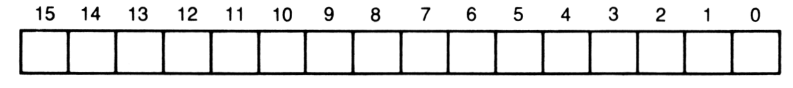


Abbildung 2.12 Numerierung der Bits

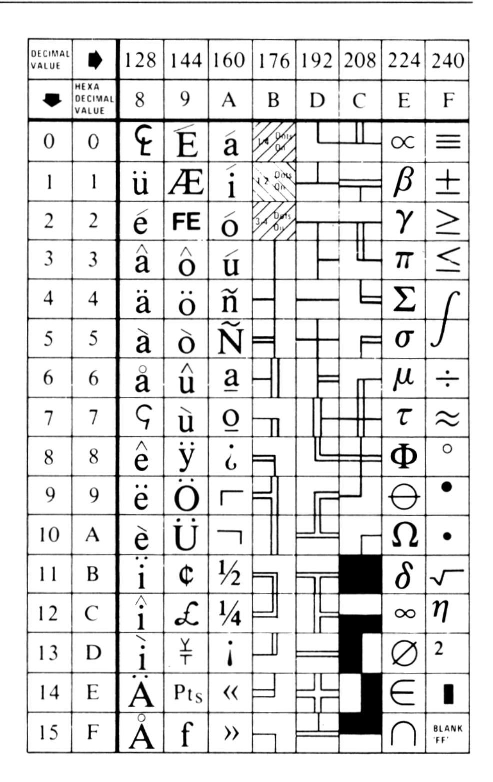
Das höchstwertige Bit in einem Wort ist Bit 15. Abbildung 2.12 zeigt ein 16-Bit-Wort mit der Numerierung der einzelnen Bits. Diese Art der Bitnumerierung finden wir auch in der gesamten Dokumentation für den IBM PC wieder.

Zeichensatz

Wie wir bereits gesehen haben, können wir jedes Byte sowohl als Binärzahl als auch als Zeichenwert betrachten. Jede der Binärzahlen zwischen 0 und 255 würde dann ein bestimmtes Zeichen darstellen. Abbildung 2.13 zeigt den Zeichensatz für den IBM PC. Die Spalten in der Tabelle entsprechen dabei den höherwertigen 4 Bits des jeweiligen Zeichens, während die Zeilen jeweils den 4 niederwertigen Bits entsprechen. So steht an der Position 41H das Zeichen A, während der Code 51H das Zeichen „ˆ" darstellt.

Der Zeichensatz des IBM PC ist eine Erweiterung des ASCII-(American Standard Code for Information Interchange) Zeichensatzes. Innerhalb des ASCII-Zeichensatzes stellen die Werte zwischen 20H und 7FH die normalen alphabetischen, numerischen und Interpunktionszeichen dar. Die Codes von 0H bis 1FH werden von Steuerzeichen belegt. Abbildung 2.14 zeigt die allgemein üblichen ASCII-Steuerzeichen. Diese Zeichen haben nur dann Bedeutung, wenn sie an den IBM-Drucker oder an einen anderen ASCII-Drucker gesendet werden. Allerdings ergeben diese Steuerzeichen, wie Abbildung 2.13 zeigt, auch graphische Symbole, sobald man sie an einen Bildschirm schickt. Der IBM PC benützt diesen Teil des ASCII-Codes für Graphik, um die zusätzlichen Fähigkeiten des Displayadapters auszuschöpfen. Da

DECIMAL VALUE →		0	16	32	48	64	80	96	112
↓	HEXA DECIMAL VALUE	0	1	2	3	4	5	6	7
0	0	BLANK (NULL)	►	BLANK (SPACE)	0	@	P	‘	p
1	1	☺	◄	!	1	A	Q	a	q
2	2	☻	↕	"	2	B	R	b	r
3	3	♥	‼	#	3	C	S	c	s
4	4	♦	¶	$	4	D	T	d	t
5	5	♣	§	%	5	E	U	e	u
6	6	♠	▬	&	6	F	V	f	v
7	7	•	↨	'	7	G	W	g	w
8	8	◘	↑	(	8	H	X	h	x
9	9	○	↓	)	9	I	Y	i	y
10	A	◙	→	*	:	J	Z	j	x
11	B	♂	←	+	;	K	[	k	{
12	C	♀	∟	,	<	L	\	l	¦
13	D	♪	↔	—	=	M	]	m	}
14	E	♫	▲	.	>	N	^	n	~
15	F	☼	▼	/	?	O	_	o	△



DECIMAL VALUE →		128	144	160	176	192	208	224	240
↓	HEXA DECIMAL VALUE	8	9	A	B	D	C	E	F
0	0	Ç	É	á	1/4 Dots On	└	╨	∝	≡
1	1	ü	Æ	í	1/2 Dots On	┴	╤	β	±
2	2	é	FE	ó	3/4 Dots On	┬	╥	γ	≥
3	3	â	ô	ú	│	├	╙	π	≤
4	4	ä	ö	ñ	┤	─	╘	Σ	⌠
5	5	à	ò	Ñ	╡	┼	╒	σ	⌡
6	6	å	û	ª	╢	╞	╓	µ	÷
7	7	ç	ù	º	╖	╟	╫	τ	≈
8	8	ê	ÿ	¿	╕	╚	╪	Φ	°
9	9	ë	Ö	⌐	╣	╔	┘	Θ	•
10	A	è	Ü	¬	║	╩	┌	Ω	·
11	B	ï	¢	½	╗	╦	█	δ	√
12	C	î	£	¼	╝	╠	▄	∞	η
13	D	ì	¥	¡	╜	═	▌	∅	²
14	E	Ä	Pts	«	╛	╬	▐	∈	▮
15	F	Å	f	»	┐	╧	▀	∩	BLANK 'FF'

Abbildung 2.13 IBM Zeichensatz aus dem TRM
(mit freundlicher Genehmigung der IBM; Copyright IBM 1981)

der Displayadapter jeden der 256 möglichen Codes direkt auf dem Bildschirm darstellen kann, gibt es keinen Grund, irgendeinen der möglichen Codes von der Verwendung auszuschließen. IBM verwendet die 32 Steuerzeichen deshalb hauptsächlich für graphische Symbole, die normalerweise nur auf dem Bildschirm und nicht auf dem Drucker ausgegeben werden. Kurz gesagt, die 32 niedrigsten Zeichen des ASCII-Codes sind Steuerzeichen, wenn sie an einen Drucker gesendet werden, ergeben aber graphische Symbole, wenn man sie an einen Bildschirm schickt.

Die Zeichen von 80H bis 0FFH sind eine Erweiterung des ASCII-Codes speziell für den IBM PC. IBM wählte diese Zeichen aus, um die Anwendungsmöglichkeiten des Rechners zu erweitern. In diesem Teil des Codes finden wir fremdsprachige Sonderzeichen, Liniengraphik und wissenschaftliche Symbole, die einen Einsatz des IBM PC auf vielen Gebieten sinnvoll werden lassen.

Irgendwann wird die Notwendigkeit auftauchen, Zeichencodes im Speicher für die spätere Verwendung durch ein Programm bereitzustellen. Ein Beispiel könnte eine Meldung an den Bediener während des Programmablaufs sein. Anstatt nun die einzelnen Codes mühsam in der Tabelle nachzusehen, können wir den Text direkt eingeben. Der Assembler ermöglicht uns das mit der DB-Anweisung.

Wert	Zeichen	Bedeutung	
0	NUL	Null	
7	BEL	Bell	
9	HT	Horizontal tab	(Tabulator)
0A	LF	Line feed	(Zeilenvorschub)
0B	VT	Vertical tab	
0C	FF	Form feed	(Formularvorschub)
0D	CR	Carriage return	(Wagenrücklauf)
0E	SO	Shift out	
0F	SI	Shift in	
11	DC1	Device control 1	
12	DC2	Device control 2	
13	DC3	Device control 3	
14	DC4	Device control 4	
18	CAN	Cancel	
1B	ESC	Escape	

Abbildung 2.14 IBM-Steuercodes

Im Operandenfeld verwenden wir dann anstelle von Zahlen einen in Hochkommas gesetzten Textstring. Der Assembler legt dann für uns die entsprechenden Codewerte, ein Byte pro Zeichen, im Speicher ab. Dazu muß man wissen, daß der Assembler nur Zeichen im Codewert zwischen 20H und 0FFH erkennen kann. Um die Steuerzeichen zwischen 0H und 1FH einzugeben, müssen wir an Stelle des in Hochkommas gesetzten Strings einen Zahlenwert verwenden. Diese Notwendigkeit ist dadurch bedingt, daß der Text in der Quelldatei einige Steuerzeichen verwendet, um Anfang und Ende der Zeilen zu markieren.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 2.15 Define Byte for ASCII Text

1                                              PAGE    ,132
2                                              TITLE   Figure 2.15 Define Byte for ASCII Text
3
4        0000  54 68 69 73 20 69               DB      'This is a message',10,13
5              73 20 61 20 6D 65
6              73 73 61 67 65 0A
7              0D
8
9                                              END
```

Abbildung 2.15 Define Byte (DB) für einen ASCII-Text

Das Beispiel in Abbildung 2.15 erzeugt 19 Datenbytes im Programm. Die ersten 17 Bytes entsprechen dabei den 17 Buchstaben in dem von Hochkommas eingeschlossenen Textstring. Das erste Byte ist also 54H, das nächste 68H, und so weiter. Die letzten beiden Bytes sind Steuerzeichen und müssen deshalb als Zahlen eingegeben werden. Diese beiden letzten Bytes der 19 Byte langen Mitteilung stellen die beiden Funktionen Wagenrücklauf und Zeilenvorschub dar. Wenn wir diese 19-Byte-Meldung an den Drucker senden, wird er den Text innerhalb der Hochkommas ausgeben. Die Steuerzeichen werden den Drucker dann veranlassen, diese Zeile zu beenden und das Papier um eine Zeile weiterzutransportieren.

Arbeitsweise des Rechners

Die folgenden Abschnitte erklären einige Grundlagen der Arbeitsweise eines Rechners. Diese Prinzipien sind wichtig für das Verstehen des 8088 und seiner Arbeitsweise, obwohl sie auch für alle anderen Rechner zutreffen. Wo es notwendig ist, werden wir speziell auf den 8088 eingehen, obwohl fast alle ausschließlich für diesen Rechner geltenden Informationen im nächsten Kapitel erscheinen werden.

Die Arbeit eines Computers besteht eigentlich nur darin, Anweisungen aus dem Speicher zu holen und sie dann auszuführen. Jeder Befehl durchläuft diesen Zweistufenprozeß. Die erste Stufe, das Lesen des Befehls aus dem Speicher, wird von einem Register des Prozessors gesteuert. Dieses Register wird als Befehlszähler bezeichnet und ist eigentlich ein Pointer. Und dieser Pointer zeigt auf die Stelle, an der wir uns im jeweils laufenden Programm gerade befinden. Die Speicherstelle, auf die dieses Register zeigt, enthält jeweils den nächsten Befehl, den der Prozessor lesen und ausführen soll. Als nächstes liest der Prozessor den Inhalt der Bytes an dieser Speicherstelle. Dann werden die Bytes vom Prozessor interpretiert und als Befehl ausgeführt. Dann erhöht der Prozessor den Befehlszähler um die Anzahl der Bytes des gerade ausgeführten Befehls. Der Befehlszähler zeigt also jetzt auf den nächstfolgenden Befehl. Und dieser Zyklus wiederholt sich solchermaßen für jeden weiteren Befehl. Der normale Ablauf eines Programms ist nämlich sequentiell, d.h. ein Befehl folgt dem nächsten.

Der Prozessor kann diesen sequentiellen Programmablauf durch die Ausführung eines Befehls unterbrechen, der einen neuen Wert in den Befehlszähler lädt. Wir nennen diese Befehle „Steuerbefehle"; sie verursachen die Fortsetzung des Programms an einem anderen Punkt des Speichers. Die meistverwendeten Steuerbefehle sind Sprung- oder Verzweigungsanweisungen. Der Sprungbefehl legt die Position der nächsten auszuführenden Anweisung fest. Eine Programmschleife ist ein typisches Beispiel für die Verwendung eines solchen Sprungbefehls. Das Beispiel in Abbildung 2.16 zeigt, in 8088-Assemblersprache, wie ein Wert in aufeinanderfolgenden Speicherzellen abgelegt wird. Der Sprungbefehl am Ende der Schleife bedingt die Wiederholung der Anweisungsfolge.

```
The IBM Personal Computer Assembler 01-01-83         PAGE    1-1
Figure 2.16   Jump Instruction

1                                            PAGE    ,132
2                                            TITLE   Figure 2.16   Jump Instruction
3         0000                       CODE    SEGMENT
4                                            ASSUME  CS:CODE
5
6         0000                       MEM     LABEL   BYTE
7
8         0000                       FIG2_16:
9         0000  2E: C6 87 0000 R 00          MOV     MEM[BX],0
10        0006  43                           INC     BX
11        0007  EB F7                        JMP     FIG2_16
12
13        0009                       CODE    ENDS
14                                           END
```

Abbildung 2.16 Sprungbefehl

Halten wir fest, daß die Sprunganweisung ein Label verwendet, in unserem Fall „FIG2-16", um den Punkt der nächsten ausführbaren Anweisung festzulegen. Dies ist zudem ein gutes Beispiel für die Fähigkeiten des Assemblers. Obwohl in der Maschinensprache die absolute Adresse der nächsten Anweisung erforderlich ist, verlangt die Assemblersprache nur ein vom Programmierer definiertes Label. Der Assembler ermittelt dann die absolute Adresse und setzt im Maschinencode die korrekte Adresse ein.

Sprunganweisungen müssen nicht, wie in unserem obigen Beispiel, unbedingt sein. Der 8088 verfügt über eine ganze Reihe von Sprunganweisungen, die nur dann ausgeführt werden, wenn ein bestimmter Bedingungscode vorliegt. Jeder Befehl kann diesen Bedingungscode verändern, wenn er vom Prozessor ausgeführt wird. Die in einer bedingten Sprunganweisung festgelegte Bedingung wird mit dem im Statusregister enthaltenen Bedingungscode verglichen. Bei Übereinstimmung verzweigt der Prozessor an die angegebene Zieladresse. Ist die Bedingung nicht erfüllt, ignoriert der Prozessor den Sprung und das Programm läuft in seiner normalen sequentiellen Weise weiter. In Abbildung 2.17 verändern wir unser oben angeführtes Beispiel. Die Schleife in unserem neuen Beispiel ist dann beendet, wenn der im BX-Register enthaltene Wert 1000 erreicht.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 2.17  Conditional Jump Instruction

1                                            PAGE    ,132
2                                            TITLE   Figure 2.17  Conditional Jump Instruction
3       0000                         CODE    SEGMENT
4                                            ASSUME  CS:CODE
5
6       0000                         MEM     LABEL   BYTE
7
8       0000                         FIG2_17:
9       0000  2E: C6 87 0000 R 00            MOV     MEM[BX],0
10      0006  43                             INC     BX
11      0007  81 FB 03E8                     CMP     BX,1000
12      000B  75 F3                          JNE     FIG2_17
13      000D  90                             NOP
14
15      000E                         CODE    ENDS
16                                           END
```

Abbildung 2.17 Bedingte Sprunganweisung

In Abbildung 2.17 fügen wir unserem Programm eine Vergleichsanweisung hinzu, die diesen Bedingungscode setzt. Die bedingte Sprunganweisung (JNE für „Jump if Not Equal") verzweigt bei erfüllter Bedingung auf „FIG2-17". Anderenfalls führt der 8088 die dem bedingten Sprung folgende Anweisung aus, in diesem Fall die Anweisung NOP. Die bedingte Sprunganweisung gestattet uns also die Überprüfung von Daten während des Programmablaufs. Aufbauend auf dieser Überprüfung kann das Programm im Weiteren verschiedene Wege einschlagen.

Unterprogramme

Eine andere Form des Sprungbefehls ist der Sprung in ein Unterprogramm. Ein Unterprogramm wird gebildet aus einer Folge von Befehlen. Diese Folge von Befehlen kann z.B. eine Funktion ausführen, die von einem Programm oft und an verschiedenen Stellen benötigt wird. Anstatt nun jedesmal diese Befehlsfolge bei Bedarf zu wiederholen, legen wir die Anweisungen in einem bestimmten Speicherbereich ab. Dieser Teil des Programms wird nun zum Unterprogramm.

Jedesmal, wenn unser Programm die im Unterprogramm enthaltene Funktion benötigt, überträgt es durch einen speziellen Sprungbefehl die Steuerung an dieses Unterprogramm. Diesen Sprung in ein Unterprogramm nennen wir Unterprogrammaufruf oder Call-Anweisung. Der Unterprogrammaufruf unterscheidet sich von einem normalen Sprungbefehl. Er stellt nämlich die Adresse der nächstfolgenden Anweisung sicher. Diese Adresse, die sogenannte Rücksprungadresse, ist der Weg zurück in unsere ursprüngliche Befehlsfolge.

Sehen wir uns nun an, wie ein Unterprogrammaufruf arbeitet. Schreiben wir z.B. ein Programm, das 32-Bit-Zahlen an verschiedenen Speicherstellen addiert. Nun gibt es aber keinen 8088-Befehl, der 32-Bit-Additionen ausführen könnte. Wir können aber ein kurzes Programm aus 8088-Befehlen schreiben, das diese Aufgabe erfüllt. Und diese Befehlsfolge wird nun ein Unterprogramm.

Für den Programmierer ist ein Unterprogramm ein Programmteil wie jeder andere. Das Unterprogramm ist also ein ganz normaler Teil eines Assemblerprogramms. Wenn wir den Hauptteil unseres Programms schreiben, werden wir des öfteren in die Verlegenheit kommen, zwei 32-Bit-Zahlen zu addieren. Anstelle nun jedesmal die explizite Befehlsfolge für die Addition niederzulegen, ist in unserem Programm ein Aufruf an ein Unterprogramm enthalten, das die Addition der 32-Bit-Zahlen übernimmt. Nach diesem Unterprogrammaufruf läuft unser Hauptprogramm ganz normal weiter. Der Unterprogrammaufruf hat also die gleiche Wirkung wie ein sehr mächtiger 8088-Befehl, denn ein einzelner Aufruf bewirkt nun eine 32-Bit-Addition.

Während des Programmablaufs wird nun bei unserem Hauptprogramm keine Addition ausgeführt, sondern die Steuerung an das Unterprogramm übergeben, das seinerseits eine 32-Bit-Addition durchführt. Der Prozessor führt also an dieser Stelle die Befehle im Unterprogramm aus, die ihrerseits die Addition bewirken. Der letzte Befehl unseres Unterprogramms ist ein spezieller Befehl, der nur in Unterprogrammen benützt wird, die sogenannte Rücksprunganweisung. Dieser Return-Befehl nimmt die Rücksprungadresse, die während der Ausführung des Unterprogrammaufrufs sichergestellt wurde, und überträgt sie in den Befehlszähler. Dies verursacht die Fortsetzung des Programms bei dem Befehl, der unmittelbar auf den Unterprogrammaufruf folgt. Ein Unterprogrammaufruf leitet also den normalen Fluß des Programms kurzfristig auf die Befehlsfolge des Unterprogramms um. Nach dem Unterprogrammaufruf läuft das Programm ganz normal weiter.

```
The IBM Personal Computer Assembler 01-01-83         PAGE    1-1
Figure 2.18  Subroutine Usage

 1                                               PAGE    ,132
 2                                               TITLE   Figure 2.18  Subroutine Usage
 3       0000                            CODE    SEGMENT
 4                                               ASSUME  CS:CODE
 5
 6       0000   E8 0008 R                A1:     CALL    SUBROUTINE
 7
 8       0003   40                       A2:     INC     AX
 9
10       0004   E8 0008 R                A3:     CALL    SUBROUTINE
11
12       0007   43                       A4:     INC     BX
13
14                                       ;----- Routine continues here . . .
15
16       0008                            SUBROUTINE      PROC    NEAR
17
18       0008   B8 0000                          MOV     AX,0
19       000B   BB 0000                          MOV     BX,0
20       000E   C3                               RET
21
22       000F                            SUBROUTINE      ENDP
23
24       000F                            CODE    ENDS
25                                               END
```

Abbildung 2.18 Verwendung von Unterprogrammen

Die bei Verwendung von Unterprogrammen benötigten Befehle sind CALL und RET. Der CALL-Befehl ist der Sprung in das Unterprogramm. Er sichert den aktuellen Wert des Befehlszählers. Dieser gesicherte Wert des Befehlszählers stellt die Rücksprungadresse dar. Der RET-Befehl liest diesen sichergestellten Wert, überträgt ihn in den Befehlszähler und übergibt die Steuerung an den solchermaßen festgelegten Befehl, der unmittelbar auf unseren Unterprogrammaufruf folgt. Das Beispiel in Abbildung 2.18 zeigt ein Unterprogramm, das von zwei verschiedenen Punkten aufgerufen wird.

Unser Beispielprogramm erreicht den CALL-Befehl beim Label A1. Der Befehl überträgt die Steuerung an die Stelle SUBROUTINE. Als weiteren Effekt des Unterprogrammaufrufs stellt der Prozessor die Adresse A2 sicher. Das Unterprogramm wird nun durchlaufen und die Anweisung RET (für RETURN) ausgeführt, was ein Rückübertragen der Adresse von A2 in den Befehlszähler bewirkt. Die Steuerung kehrt solchermaßen zurück zum Hauptprogramm. Später in unserem Hauptprogramm wird ein Unterprogrammaufruf bei A3 ausgeführt, was für ein weiteres Mal das Ausführen des Unterprogramms bewirkt. In diesem Fall stellt der Prozessor den Adresswert von A4 sicher. Nachdem wir nun unser Unterprogramm wiederum durchlaufen haben, kehren wir also an die Adresse A4 zurück. Halten wir dabei fest, daß das gleiche Unterprogramm zweimal ausgeführt wurde. Beim erstenmal erfolgte der Rücksprung aus dem Unterprogramm nach A2, beim zweitenmal nach A4. Die Stärke eines Unterprogramms liegt also in seiner Fähigkeit, von vielen verschiedenen Stellen aus aufgerufen werden zu können, und immer wieder korrekt an die aufrufende Stelle zurückzukehren.

Wo wird nun diese Rückkehradresse während der Ausführung des Unterprogramms gespeichert? Dafür gibt es viele Möglichkeiten, der 8088 jedenfalls speichert den Wert im Stack.

Stack

Ein Stack ist eine Datenstruktur, die für die zeitweilige Speicherung von Informationen verwendet wird. Ein Programm kann im Stack Informationen speichern (PUSH) oder sie wieder aus dem Stack auslesen (POP). Die Struktur des Stack bedingt eine spezielle Art der Datenspeicherung. Es werden immer die Daten aus dem Stack ausgelesen, die zuletzt hineingeschrieben wurden. Die Bezeichnung für dieses Vorgehen lautet LIFO, für last in, first out. Wenn wir zwei Daten im Stack ablegen, zuerst A, dann B, und den Stack dann wieder zurücklesen, erhalten wir zuerst B. Beim zweiten Auslesen erhalten wir dann A. Die Information wird aus dem Stack also genau in entgegengesetzter Reihenfolge ausgelesen, als sie ursprünglich in den Stack geschrieben wurde. Wir können den Stack als Gegenstück zu einer Warteschlange verstehen. Eine Warteschlange entspricht einer normalen Schlange, wie man sie vor einem Postschalter oder vor einer Bushaltestelle sehen kann. Eine Warteschlange ist eine Datenstruktur, die wir als first in, first out (FIFO), bezeichnen. Die erste Person, die in einer solchen Warteschlange steht, ist auch die erste Person, die sie wieder verläßt. Eine Warteschlange und ein Stack sind also extrem verschieden.

Auf dem Rechner wird der Stack verwirklicht als reservierter Speicherbereich mit einem Zeiger, den wir als Stackpointer bezeichnen. Für unser Programm ist es wichtig zu wissen, daß der Stackpointer immer auf den zuletzt gemachten Eintrag im Stack zeigt. Im Gegensatz dazu bewegt sich z.B. in unserem Postamt jedes Element der Warteschlange weiter, wenn sich die Schlange vorwärts bewegt. In unserem Rechner ist es viel einfacher, einen Zeiger auf die Daten zu verwenden und nur den Zeiger zu verändern, wenn wir neue Daten hinzufügen oder alte wieder wegnehmen. Der Stackpointer wird also in Abhängigkeit von POP- und PUSH-Befehlen in seinem Wert erhöht oder erniedrigt.

Abbildung 2.19 zeigt ein Beispiel. In Reihe (a) sehen wir den Stack, nachdem die Werte A, B und C (in dieser Reihenfolge) in den Stack geschrieben wurden. Der Stackpointer zeigt auf die aktuelle Spitze des Stacks (Top of Stack, TOS), in unserem Fall das Element C. In Reihe (b) wird dem Stack ein weiterer Wert, D, hinzugefügt. Der PUSH-Befehl erniedrigt den Stackpointer (SP), der nun auf die neue TOS, also den Wert D, zeigt. Der Stackpointer enthält so immer die Adresse des zuletzt dem Stack hinzugefügten Wertes.

Abbildung 2.19, Reihe (c), zeigt den Stack nach einem POP-Befehl. Der POP-Befehl hat den Wert D aus dem Stack entfernt. Er transportiert den aus dem Stack entnommenen Wert an einen von uns bestimmten Platz. Wäre der Befehl in Reihe (c) z.B. POP AX, würde der Prozessor den Wert D aus dem Stack in das Register AX transportieren (einer von mehreren Punkten, die wir im nächsten Kapitel in Einzelheiten besprechen werden). Der POP-Befehl erhöht den Stackpointer. Er zeigt nun wiederum auf das neue TOS, in unserem Fall den Wert C. Halten wir fest, daß Werte aus dem Stack in der Art LIFO entnommen werden. Das letzte in den Stack geschobene Element war D, und das erste aus ihm wieder herausgelesene ist wiederum D.

Halten wir außerdem fest, daß der Wert D im Speicher erhalten bleibt, daß er aber nicht länger Teil des Stacks ist. Per Definition endet der Stack an der durch den Stackpointer bestimmten Adresse. Die Spitze des Stacks liegt also jetzt unterhalb dem Wert D.

Abbildung 2.19, Reihe (d), zeigt, was mit dem Wert D geschieht, wenn wir dem Stack einen neuen Wert E hinzufügen. Der Wert E hat nämlich den Wert D überschrieben und ist nun seinerseits die neue Spitze des Stacks. Die Moral der Geschichte ist, daß Werte, die wir bereits aus dem Stack gelesen haben, sich immer noch dort befinden können, aber darauf verlassen können wir uns nicht.

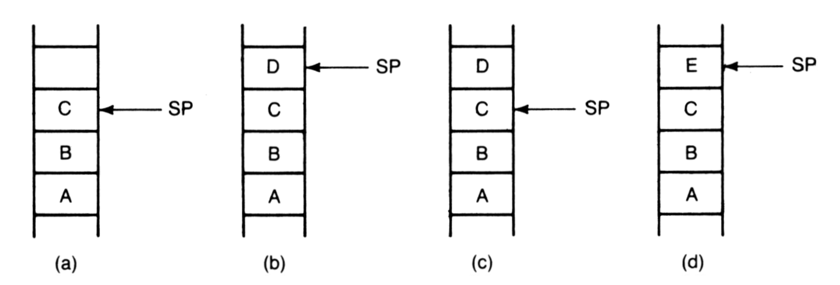


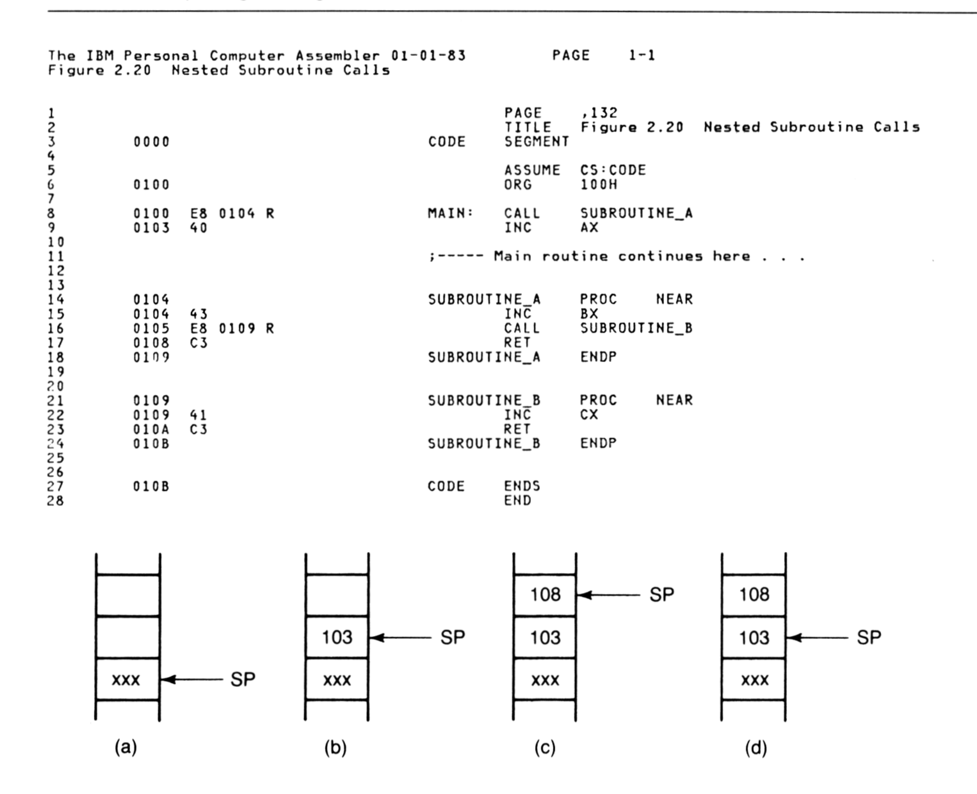
Abbildung 2.19 Beispiel für den Stack

In den bisherigen Beispielen haben wir gezeigt, wie der 8088 den Stack benützt. Der Stackpointer zeigt dabei immer auf die aktuelle Spitze des Stacks. PUSH-Befehle erniedrigen den Stackpointer, POP-Befehle erhöhen ihn. Der Stack wächst dabei immer in Richtung auf die niedrigere Speicheradresse. Der Beginn des Stacks befindet sich immer an einer höheren Speicheradresse als die Spitze des Stacks. Wenn wir uns ein Bild zeichnen mit der niedrigsten Speicheradresse oben, wie in Abbildung 2.19, dann befindet sich auch die Spitze des Stacks an der Spitze des Bildes.

Wir besprechen den Stack unter anderem deshalb, weil er auch für die Rückkehradressen der Unterprogramme von Bedeutung ist. Wie geschieht dies?

Jeder CALL-Befehl verursacht einen PUSH auf den Stack. Der CALL-Befehl sichert auf diese Weise die Rücksprungadresse im Stack. Der RET-Befehl macht einen POP auf den Stack, um solchermaßen die Rücksprungadresse wieder in den Befehlszähler laden zu können. Und der 8088 benützt den Stack für die Speicherung der Rücksprungadressen, um ein Verschachteln von Unterprogrammen zu ermöglichen. Warum Verschachteln? Abbildung 2.20 zeigt ein Beispiel von ineinander verschachtelten Unterprogrammen.

Abbildung 2.20 ist ein völlig sinnloses Programm, das wir nur verwenden, um ein Beispiel für die Verschachtelung von Unterprogrammen zu geben. Reihe (a) zeigt uns den Stack vor der Programmausführung. Wird das Hauptprogramm nun ausgeführt, ruft es das Unterprogramm SUBROUTINE-A auf. Nun speichert der Prozessor die Rückkehradresse im Stack. Reihe (b) zeigt, wie diese Rückkehradresse, 103, in den Stack geschoben wird. SUBROUTINE-A ruft nun seinerseits ein Unter-



```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 2.20   Nested Subroutine Calls

 1                                               PAGE    ,132
 2                                               TITLE   Figure 2.20  Nested Subroutine Calls
 3       0000                            CODE    SEGMENT
 4
 5                                               ASSUME  CS:CODE
 6       0100                                    ORG     100H
 7
 8       0100  E8 0104 R                 MAIN:   CALL    SUBROUTINE_A
 9       0103  40                                INC     AX
10
11                                       ;----- Main routine continues here . . .
12
13
14       0104                            SUBROUTINE_A    PROC    NEAR
15       0104  43                                INC     BX
16       0105  E8 0109 R                         CALL    SUBROUTINE_B
17       0108  C3                                RET
18       0109                            SUBROUTINE_A    ENDP
19
20
21       0109                            SUBROUTINE_B    PROC    NEAR
22       0109  41                                INC     CX
23       010A  C3                                RET
24       010B                            SUBROUTINE_B    ENDP
25
26
27       010B                            CODE    ENDS
28                                               END
```


Abbildung 2.20 Verschachtelte Unterprogrammaufrufe

programm mit Namen SUBROUTINE-B auf. Dieser CALL-Befehl speichert nun seine Rückkehradresse in den Stack, wie wir in Reihe (c) sehen. Der Wert 108 ist also die Rücksprungadresse in das Unterprogramm SUBROUTINE-A. Wenn das Unterprogramm SUBROUTINE-B beendet ist, liest die RET-Anweisung ihrerseits den Wert 108 aus dem Stack, wie in Reihe (d) dargestellt. Der Prozessor speichert nun diesen Wert, wie von der RET-Anweisung verlangt, im Befehlszähler. Wie wir aus der Assemblerliste ersehen können, befindet sich die Stelle 108 im Unterprogramm SUBROUTINE-A und stellt genau die Adresse dar, die auf den Aufruf von Unterprogramm SUBROUTINE-B folgt. Nun endet seinerseits das Unterprogramm SUBROUTINE-A. Die RET-Anweisung liest den Wert 103 aus dem Stack und überträgt ihn in den Befehlszähler. Die Stelle 103 befindet sich nun wieder im Hauptprogramm, und zwar unmittelbar nach dem Aufruf von Unterprogramm SUBROUTINE-A.

Das Wichtigste am Beispiel in Abbildung 2.20 ist nun das Ineinanderverschachteln der Unterprogramme. Jedes Unterprogramm kann ein beliebiges anderes Unterprogramm aufrufen. Der RET-Befehl sorgt jeweils wieder für die korrekte Rückkehr in

das aufrufende Programm. Die einzige Grenze für die Tiefe der Verschachtelung (d.h., wieviele Unterprogramme jeweils andere Unterprogramme aufrufen können) ist die Größe des Stacks. Solange im Stack also noch Platz für eine weitere Rückkehradresse ist, kann ein weiteres Unterprogramm aufgerufen werden. Die LIFO-Struktur des Stacks bewahrt dabei die korrekte Abfolge der Rückkehradressen.

Das Beispiel in Abbildung 2.20 zeigt außerdem die Anwendung einer weiteren Assembler-Pseudoanweisung, nämlich PROC. Der Assembler benützt die PROC-Anweisung, um Unterprogramme zu definieren. Wie wir später sehen werden, muß der Assembler wissen, wie weit ein Unterprogramm vom Punkt des Aufrufs entfernt ist, und wie die Rückkehr zum aufrufenden Programm abgewickelt werden soll. Der Operand NEAR teilt dem Assembler dabei mit, daß das Unterprogramm sich innerhalb der einfachen Reichweite zum aufrufenden Programm befindet. Wir werden später noch einmal auf die PROC-Anweisung zurückkommen, wenn wir die Anwendung der CALL- und JMP-Befehle besprechen.

Unterbrechungen (Interrupts)

Der Unterbrechungsmechanismus ist ein wichtiger Teil eines jeden Rechners. Wie wir sehen werden, ist er auch für den IBM PC von großer Bedeutung. Die Unterbrechungsstruktur sieht dabei ein wirksames Mittel für die Kommunikation der Ausgabegeräte mit dem eigentlichen Prozessor vor. Wir interessieren uns für Unterbrechungen, weil ihre Bearbeitung eines der Gebiete ist, das nur mit der Assemblersprache behandelt werden kann. Höhere Programmiersprachen verfügen nämlich nicht über die Mittel, Unterbrechungen auf Maschinenebene zu bearbeiten.

Externe Geräte verwenden normalerweise Unterbrechungen. Diese Unterbrechungen veranlassen den Prozessor, seine augenblickliche Tätigkeit einzustellen und direkt auf das externe Gerät zu antworten. Beim IBM PC sendet z.B. die Tastatur ein Unterbrechungssignal jedesmal wenn eine Taste gedrückt wird. Diese Tastaturunterbrechung veranlaßt den Prozessor, seine augenblickliche Tätigkeit zu unterbrechen und das Zeichen zu lesen, das gerade auf der Tastatur anliegt.

Die Bezeichnung Unterbrechung ist beinahe selbsterklärend. Das Unterbrechungssignal unterbricht nämlich die augenblickliche Tätigkeit des Prozessors. Die Unterbrechung ist deswegen so praktisch, weil sie den Prozessor davon befreit, dauernd die externen Geräte zu überwachen. Würde die Tastatur z.B. keine Unterbrechung abgeben, so müßte der Prozessor dauernd die Tastatur überprüfen, um festzustellen, ob irgendeine Taste gedrückt wurde. Und jedes für den Rechner geschriebene Programm müßte nun das gleiche tun, und so sehr oft die Tastatur im Programm überprüfen. Doch eine Unterbrechung entledigt uns dieser Arbeit. Das Programm kann nun ablaufen ohne die Tastatur zu überprüfen. Jedesmal, wenn an der Tastatur eine Information anliegt, teilt sie dies dem Prozessor mit. Und wenn der Prozessor dann die von der Tastatur ausgelöste Unterbrechung bearbeitet hat, kann er ganz normal den Programmablauf fortführen.

Der 8088 behandelt Unterbrechungen in ähnlicher Weise wie Unterprogramme. Wenn eine Unterbrechung auftritt, so kann sie nicht einfach den Prozessor mitten in der Abarbeitung eines Befehls unterbrechen. Der 8088 beendet immer zuerst den gerade ablaufenden Befehl. Der nächstfolgende Befehl wird allerdings bereits nicht mehr ausgeführt. Stattdessen tut der Prozessor so, wie wenn der nächste Befehl ein Unterprogramm wäre. Der Prozessor stellt also die Adresse des nächsten Befehles im Stack sicher und springt in ein spezielles Unterprogramm zur Unterbrechungsbehandlung. Dieses Unterprogramm, bekannt als Unterbrechungsroutine, enthält genau die Anweisungen, die nötig sind, das Gerät zu bedienen, das die Unterbrechung auslöst. Im Falle der Tastatur liest die Unterbrechungsroutine ganz einfach das Zeichen, das an der Tastatur anliegt, und bewahrt es zur späteren Benutzung auf. Nachdem die Unterbrechungsroutine die Bearbeitung des jeweiligen Geräts abgeschlossen hat, kehrt sie zum Punkt der Unterbrechung zurück. Der Prozessor liest dann die Rückkehradresse wieder aus dem Stack und führt das Programm ganz normal weiter wie wenn nichts geschehen wäre.

Da hauptsächlich externe Geräte Unterbrechungen auslösen, kann eine Unterbrechung jederzeit und überall in einem Programm stattfinden. Das heißt auch, daß Programme keine besonderen Vorkehrungen für den Fall von Unterbrechungen treffen können. So kann ein Programm z.B. nicht vorhersehen, wann ein bestimmter Benutzer ein Zeichen auf der Tastatur eingeben wird. Das bedeutet, daß die Unterbrechungsroutine keinesfalls die vom Programm benützten Werte verändern darf. Sollte eine Unterbrechungsroutine irgendeinen vom Programm benützten Wert verändern, so können wir beinahe sicher sein, daß das Programm nicht mehr korrekt weiterläuft, sobald die Unterbrechung abgearbeitet ist.

Als Teil der Unterbrechungsbehandlung rettet der 8088 einige Programmwerte automatisch in den Stack. Es liegt nämlich in der Verantwortung der Unterbrechungsroutine, alle jene Werte vorher zu sichern, die während der Unterbrechungsbehandlung modifiziert werden könnten. Normalerweise werden diese Werte im Stack gespeichert. Später, bevor die Steuerung wieder an das unterbrochene Programm zurückgeht, muß die Unterbrechungsroutine diese Orginalwerte wiederherstellen. Die Tatsache, daß eine Unterbrechung auftrat, muß für das ablaufende Programm völlig unsichtbar bleiben.

Da es viele Geräte geben kann, die Unterbrechungen an den Prozessor senden, verfügt der 8088 über einen Mechanismus, die Unterbrechungen in einem Vektor darzustellen. D.h., daß der 8088 feststellt, welches Gerät die Unterbrechung hervorgerufen hat, und dann die Steuerung an die entsprechende Unterbrechungsroutine für das jeweilige Gerät überträgt. Der Prozessor legt dabei automatisch die Reihenfolge der Unterbrechungsanforderungen fest. Die Unterbrechungsroutine muß also nicht erst feststellen, von welchem Gerät die Unterbrechungsanforderung ausging, wenn sie eine solche bearbeiten soll. Dies erhöht die Geschwindigkeit der Abarbeitung von Unterbrechungen und macht außerdem das Programmieren von Unterbrechungsroutinen einfacher.

Es gibt zudem einige Programmteile, die niemals unterbrochen werden dürfen. So kann es z.B. sein, daß ein bestimmter Programmteil extrem schnell ablaufen muß,

um eine bestimmte Aufgabe zu erfüllen. Oder ein Programm könnte z.B. Daten verändern, die später von einer Unterbrechungsroutine bearbeitet werden müssen. In beiden Fällen muß das Programm verhindern können, daß Unterbrechungen auftreten. D.h., das Programm muß dafür sorgen, daß in diesen kritischen Augenblicken keine Unterbrechungen auftreten. Nach diesen Abschnitten allerdings muß das Programm Unterbrechungen wieder zulassen. Ein Programm sollte das Unterbrechungssystem nicht zu lange abschalten, sonst könnte es nämlich Schwierigkeiten mit Geräten geben, die auf Unterbrechungen angewiesen sind. Liest z.B. die Tastaturunterbrechungsroutine das gerade anliegende Zeichen nicht rechtzeitig, so könnte es geschehen, daß ein weiteres gerade eingegebenes Zeichen verlorengeht. Der 8088 verfügt über die Möglichkeit, alle externen Unterbrechungen abzuschalten. Der IBM PC verfügt außerdem über eine zusätzliche Möglichkeit, auszuwählen, welches Gerät eine Unterbrechung erzeugen kann und welches nicht. Ein Programm könnte diese Fähigkeit dazu verwenden, um festzulegen, welche kritischen Geräte Unterbrechungen erzeugen dürfen, und ihnen diese auch gestatten, während weniger empfindliche Geräte von Unterbrechungsanforderungen ausgeschlossen werden. Wir werden auf die Möglichkeit des Abschaltens von Unterbrechungen in späteren Kapiteln genauer eingehen.

3 Der Mikroprozessor 8088

Der 8088 ist Teil einer Familie von Mikroprozessoren, die den 8088, den 8086 und den Arithmetikprozessor 8087 umfaßt. Der 8086 ist dabei der größere Bruder des 8088. Er hat den völlig gleichen Befehlssatz, verfügt aber über eine größere Leistungsfähigkeit. Dies beruht auf der Tatsache, daß der Datenbus, der primäre Datenweg zwischen Prozessor und Speicher, doppelt so breit ist wie der des 8088. Der 16 Bit breite Datenbus erlaubt es, in einer einzigen Operation ein komplettes Speicherwort in den Prozessor zu übertragen. Der 8 Bit breite Datenbus des 8088 dagegen benötigt für den gleichen Vorgang zwei Speicherzyklen. In praktisch allen anderen Punkten sind die beiden Prozessoren identisch. Ein für den 8088 geschriebenes Programm läuft absolut gleich auf dem 8086 ab, wahrscheinlich allerdings etwas schneller. Es dürfte übrigens sehr schwierig sein, ein Programm zu entwickeln, das in der Lage ist, ohne externe Zeitreferenzen zu unterscheiden ob es auf dem 8088 oder auf dem 8086 abläuft.

Der 8087 als Arithmetikprozessor erweitert sowohl den Befehlssatz des 8086 als auch den des 8088. Dieser sogenannte Coprozessor teilt sich dabei die Arbeit mit dem 8086 bzw. 8088 als Hauptprozessor. Der Coprozessor bearbeitet nur Befehle, die etwas mit Gleitpunktarithmetik bzw. doppelt genauer Arithmetik zu tun haben. Der 8087 kann im übrigen auch in den IBM PC eingesetzt werden. In Kapitel 7 werden wir den 8087 genauer besprechen.

Beschreibung des 8088

Um den 8088 zu verstehen und um zu lernen, wie man ihn programmiert, beginnen wir mit einer Beschreibung der internen Fähigkeiten des Prozessors. Innerhalb des Prozessors gibt es bestimmte Speicherstellen, die wir als Register bezeichnen. Diese Register können zur Speicherung von Daten, aber auch zur Speicherung von Adressen verwendet werden. Da sich die Register innerhalb des Prozessors selbst befinden, hat der Prozessor einen extrem schnellen Zugriff auf diese Daten, wesentlich schneller als wenn sich die Daten im Speicher befinden würden. Benötigt ein Programm also einen sehr schnellen Zugriff auf bestimmte Werte, so wird es die Geschwindigkeit der Abarbeitung erheblich erhöhen, wenn wir diese Werte in einem Register speichern.

Der Registersatz des 8088 besteht aus verschiedenen Gruppen von Registern. Abbildung 3.1 zeigt die Register des 8088 und ihre Gruppierung.

Allgemeine Register

Die erste Registergruppe umfaßt die Register, die hauptsächlich für Rechenzwecke eingesetzt werden. Diese Register sind alle 16 Bits breit, doch besteht die Möglichkeit, sowohl die niederwertigen als auch die höherwertigen 8 Bits jeweils getrennt anzusprechen. So ist z.B. das Register AX 16 Bits breit. Ein Programm kann die

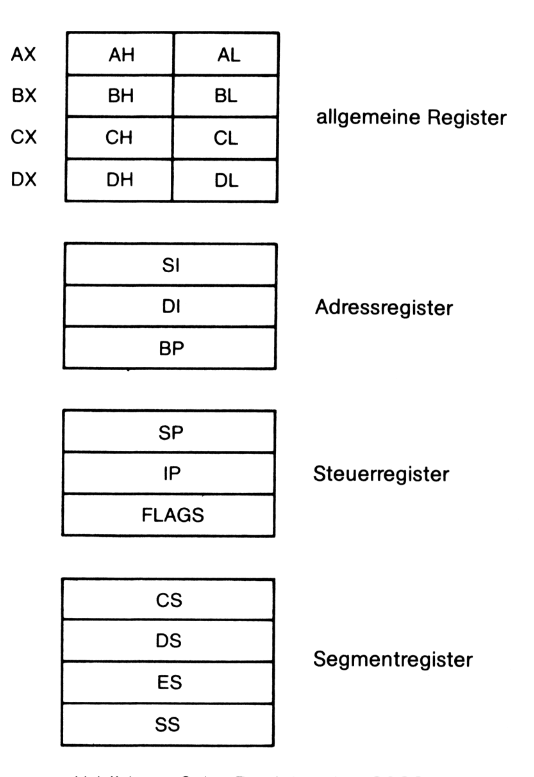


Abbildung 3.1 Register des 8088

höherwertigen 8 Bits des Registers AX unter der Bezeichnung AH ansprechen, während die niederwertigen 8 Bits des Registers AX unter der Bezeichnung AL ansprechbar sind. Dies trifft auch für die Register BX, CX, und DX zu. Ein Programm kann also diese Gruppe von Registern als entweder vier 16-Bit-Register, acht 8-Bit-Register, oder eine beliebige Kombination von 8- und 16-Bit-Registern ansprechen.

Der Hauptzweck dieser allgemeinen Register ist es, Speicheroperanden zu beinhalten. Man könnte die Register auch charakterisieren durch ihre Fähigkeit, entweder Worte oder Bytes zu enthalten. Allerdings können einige Register auch eine besondere Bedeutung haben, die ihnen durch bestimmte Befehle zugewiesen wird. Oder aber sie verfügen über spezielle Fähigkeiten, die sie über die übrigen Register herausheben. Im weiteren werden wir auf einige der speziellen Verwendungen der allgemeinen Register eingehen.

Das Register AX entspricht dem Akkumulator früherer Prozessoren. Während der 8088 besonders in Hinsicht auf Rechenoperationen wesentlich leistungsfähiger ist als frühere Prozessoren wie z.B. der 8080, verfügt er zusätzlich über einige Sonderfunktionen. Der Hersteller, Intel, optimierte den Befehlssatz des 8088 für bestimmte Befehle auf dem Weg über die zusätzliche Verwendung des AX-Registers. So sind

z.B. Direktbefehle Befehle, in denen einer der Operanden einen Wert darstellt, der bereits fest in der Anweisung enthalten ist. Diese Direktbefehle benötigen, wenn sie das AX- oder AL-Register (den 16-Bit- oder 8-Bit-Akkumulator) benutzen, weniger Speicherplatz als wenn man sie auf ein beliebiges anderes Register anwendet. Diese geringere Befehlslänge ermöglicht ein kürzeres Programm und damit eine schnellere Abarbeitung.

Das BX-Register dient als Adress- und Rechenregister. Wird es als 16-Bit-Register verwendet, kann es dazu dienen, einen Teil der Adresse eines Operanden darzustellen. Im nächsten Abschnitt werden wir die verschiedenen Adressiermöglichkeiten des 8088 darstellen.

Im Befehlssatz des 8088 wird außerdem das CX-Register als Zähler für einige bestimmte Befehle verwendet. Diese Befehle verwenden den im CX-Register gespeicherten Wert als Zähler, um die Anzahl der Iterationen durch einen Befehl oder einen Programmteil festzulegen.

Das DX-Register dient als Erweiterung des Akkumulators für doppelt genaue Multiplikations- und Divisionsbefehle. In diesem Fall werden sowohl das AX- als auch das DX-Register verwendet.

Adressregister

Der 8088 verfügt über vier verschiedene Register, die zur Adressierung von Operanden verwendet werden können. Eines dieser Register ist zugleich auch ein allgemeines Register — das BX- oder Basisregister. Die anderen drei Register sind Base Pointer (BP), Source Index (SI) und Destination Index (DI). Ein Programm kann sowohl das BP-, das SI-, oder DI-Register als 16-Bit-Operanden verwenden, doch die einzelnen Bytes dieser Register sind nicht ansprechbar. Der Hauptzweck dieser Register ist es, 16-Bit-Werte für Adressierungszwecke aufzunehmen.

Jeder Befehl des 8088 enthält irgendeine bestimmte Operation, die der Prozessor ausführen soll. Die jeweils gewünschte Operation kann dabei keinen, einen, oder zwei Operanden benötigen. So benötigt z.B. das Einschalten von Unterbrechungen über die Anweisung STI (Set Interrupt) keine Operanden. Die Anweisung INC (Increment) benötigt dagegen einen Operanden, ein Register oder eine Speicherstelle, deren Inhalt erhöht werden soll. Der Additionsbefehl benötigt zwei Operanden, nämlich die beiden Werte, die addiert werden sollen. Während es einige wenige Anweisungen gibt, die die Adresse des Operanden implizit beinhalten, erlauben es die meisten Anweisungen dem Programmierer, entweder ein Register oder aber eine Speicherstelle als Operand auszuwählen. Soll ein Register als Operand verwendet werden, so hat der Programmierer nur den Namen des Registers anzugeben. Darüberhinaus gibt es eine ganze Anzahl weiterer Möglichkeiten, wie der Programmierer eine Speicherstelle als Operand definieren kann.

Die Anweisung INC kann dabei als gutes Beispiel dienen. Sie benötigt nur einen einzigen Operanden. Abbildung 3.2 ist eine Assemblerliste, die verschiedene Möglichkeiten der Anweisung INC zeigt. Der erste INC-Befehl verwendet das Register BX als

```
The IBM Personal Computer Assembler 12-11-82        PAGE    1-1
Figure 3.2 Operand Addressing


 1                                          PAGE    ,132
 2                                          TITLE   Figure 3.2 Operand Addressing
 3
 4      0000                      CODE      SEGMENT
 5
 6                                          ASSUME  CS:CODE,DS:CODE,SS:CODE
 7
 8      0123                                ORG     123H
 9      0123   ????               OPND      DW      ?
10
11      0200                                ORG     200H
12
13      0200   43                           INC     BX              ; Increment the register
14      0201   FF 06 0123 R                 INC     OPND            ; Increment the word in memory
15      0205   FF 07                        INC     WORD PTR [BX]   ; Increment the memory location
16
17      0207   FF 87 0123 R                 INC     [OPND+BX]       ; Displacement plus index
18      020B   FF 84 0123 R                 INC     [OPND+SI]
19      020F   FF 84 0123 R                 INC     OPND[SI]        ; Different method, same results
20      0213   FF 85 0123 R                 INC     OPND + [DI]
21      0217   FF 86 0123 R                 INC     [OPND] + [BP]
22
23      021B   FF 00                        INC     WORD PTR [BX + SI]      ; Base plus index
24      021D   FF 03                        INC     WORD PTR [BP] + [DI]
25
26      021F   FF 81 0123 R                 INC     [OPND + BX + DI]        ; Base + index + displacement
27      0223   FF 82 0123 R                 INC     OPND[BP][SI]
28
29      0227                      CODE      ENDS
30                                          END
```

Abbildung 3.2 Operandenadressierung

Operand. Halten wir dabei fest, daß im Operandenfeld nur BX erscheint. Die restlichen Anweisungen in unseren Beispielen benützen Speicherstellen als Operanden. Obwohl auch dort an einigen Stellen der Name BX erscheint, ist er kein Operand. Das Register wird in diesem Fall nur verwendet, um die Position des Operanden festzulegen.

Direkte Adressierung

Die einfachste Methode, eine Speicherstelle als Operand festzulegen, ist dieser Speicherstelle einen Namen zu geben. Im weiteren Verlauf des Programms wird dann immer dieser Name verwendet, um die Speicherstelle anzusprechen. Diese Methode wird z.B. in der folgenden Anweisung

```
INC     OPND
```

verwendet. Unser Beispielprogramm verwendet dabei den Operanden OPND als Speicherstelle, die ein Wort lang ist. Die Definition erfolgt durch die Assembleranweisung

```
OPND    DW  ?
```

Wenn unser Programm nun die Bezeichnung OPND als Operanden verwendet, ersetzt der Assembler jedesmal im erzeugten Befehl den Namen OPND durch die aktuelle Adresse dieses Operanden. In unserem Beispiel können wir dann als Adresse den Wert 0123 als Teil des Befehls feststellen. Da der Befehl auf diese Weise die vollständige Adresse des Operanden direkt in sich enthält, wird diese Methode der Adressierung als direktes Adressieren bezeichnet.

Adressberechnung

Die direkte Adressierung eines Operanden im Speicher ist ein Vorgang von bestechender Einfachheit, doch gibt es viele Fälle, in denen ein Programm die aktuelle Speicherstelle erst berechnen muß. Der einfachste Fall dabei ist ein Vektor, ein eindimensionales Array. In einem FORTRAN-Programm können wir eine solche Struktur ganz einfach durch die folgende Anweisung erzeugen:

```
DIMENSION    OPND(20)
```

Den gleichen Zweck erfüllt auch eine ähnliche Deklaration in einer beliebigen anderen höheren Programmiersprache. Während der Ausführung spricht dann das Programm die einzelnen Elemente entsprechend einem Indexwert an, wie z.B. OPND(5). Dieses Programm in Assemblersprache zu schreiben, würde bedeuten, daß der Programmierer zu berechnen hätte, wo sich der fünfte Eintrag im Datenbereich befindet, der dem Operanden OPND zugewiesen wurde. Das Programm könnte dann diesen Wert für eine direkte Adressierung verwenden. Dagegen gibt es für den Assemblerprogrammierer keine Möglichkeit mehr, festzustellen, wie die korrekte direkte Adresse aussehen würde, wenn der Ausdruck OPND(I) lautet und zudem das Programm erst zur Ablaufzeit den Wert der Variablen I festlegen kann. Die Adresse muß also während der Laufzeit des Programms ermittelt werden.

Der Befehlssatz des 8088 bietet uns mehrere Möglichkeiten, die effektive Adresse (EA) eines Operanden zu bestimmen. Diese verschiedenen Wege der Adressberechnung bezeichnen wir als Adressierungsarten. Sie sind vorgesehen, um die Bestimmung der effektiven Adresse eines Operanden zu erleichtern. Durch die Wahl einer geeigneten Adressierungsart kann der Programmierer den Aufwand zur Adressberechnung in seinem Programm minimieren.

So lautet z.B. die Formel zum Festlegen der Adresse des I-ten Elements im Array OPND

Effektive Adresse = Basisadresse von OPND + (I × Länge)

wobei in diesem Fall „Länge" die Länge jedes einzelnen Eintrags im Array bedeutet. Im vorliegenden Fall ist OPND ein Wortarray, so daß jedes Element 2 Bytes lang wäre. Die Formel lautet dann

EA = Basisadresse + (I × 2)

Die Mindestvoraussetzung zum Durchführen der Adressberechnung ist ein Register, das die Adresse des Operanden enthalten muß. Das Programm kann dann die effektive Adresse berechnen, wobei das Resultat in einem anderen Register abgelegt wird. Der Befehl INC spricht dann das Register an, das die Adresse enthält, anstelle die Adresse direkt über den Operanden zu ermitteln.

Das Programm kann jedes der vier Adressregister zum Speichern von Operandenadressen verwenden. So addiert das Programm im folgenden Beispiel 2 × I auf die Basisadresse und speichert das Ergebnis im BX-Register. Der entsprechende Vektoreintrag könnte dann mit der folgenden Anweisung erhöht werden:

```
INC    WORD PTR [BX]
```

Dabei teilt der Ausdruck [BX] dem Assembler mit, daß das BX-Register die Adresse des Operanden enthält, und nicht der Operand selbst ist. Die eckigen Klammern, die den Wert umschließen, teilen dem Assembler also mit, daß es sich dabei um die Adresse einer Speicherstelle handelt. Der Assembler benötigt den zweiten Teil des Operanden, nämlich WORD PTR, um zu erkennen, daß es sich um eine Wortvariable handelt. Wir werden den Operator PTR später genauer besprechen.

Adressierung über Basis + Displacement

Nachdem die Berechnung der Adresse eines Operanden, bestehend aus Basis und Index, sehr häufig ist, verfügt der 8088 über einen Adressierungsmodus, der die Indexaddition automatisch ausführt. Anstelle nun die ganze Berechnung selbst durchzuführen, kann ein Programm den Wert von 2 × I feststellen und das Ergebnis in das BX-Register abliefern. Der nachfolgende Befehl

```
INC   [OPND + BX]
```

errechnet die effektive Adresse, indem er die Basisadresse von OPND auf den Indexwert addiert, der im BX-Register enthalten ist. Solchermaßen sprechen wir die gleiche Speicherstelle an wie im vorangegangenen Beispiel, nur mit wesentlich weniger Instruktionen. Halten wir fest, daß der Assembler in diesem Fall die Angabe WORD PTR nicht benötigt. Dies rührt daher, daß dem Assembler bekannt ist, daß es sich bei OPND um eine Wortvariable handelt. Der Assembler benötigt nämlich die Angabe PTR nur, wenn er nicht entscheiden kann um welchen Typ von Operanden es sich handelt.

In einem Programm können wir jedes der vier Adressregister als Indexregister zusammen mit einer Basisadresse verwenden. Abbildung 3.2 zeigt einige der möglichen Variationen und Kombinationen von Basisadressen und Index. Wir sehen dabei, daß der Assembler mehrere verschiedene Wege der Adressdarstellung im Programm akzeptiert. Die fünf verschiedenen Anweisungen in Abbildung 3.2 addieren alle die Basisadresse von OPND auf das genannte Indexregister.

Übrigens muß nicht immer das Register den Index und der Befehl die Basisadresse enthalten. Und in der Tat, nachdem das BX-Register als Basisregister bezeichnet wird, können wir durchaus auch den umgekehrten Weg gehen. Als Beispiel dafür nehmen wir einmal an, daß unser Programm viele verschiedene Vektoren anspricht, die alle gleich lang sind und alle gleich große Elemente enthalten. Das Notenbuch eines Lehrers z.B., das für jede Schularbeit einen eigenen Ergebnisvektor enthält, könnte eine solche Datenstruktur sein. Ein Programm, das z.B. die Noten für den fünften Schüler einer Klasse für die I-te Schulaufgabe ermitteln will, kennt also bereits den Indexwert (5), während die Basisadresse erst während des Ablaufs des Programms ermittelt werden kann (der Vektor für die entsprechende Schularbeit).

Das Indexregister kann also entweder die Basisadresse des Vektors, oder aber den Indexwert des entsprechenden Wertes innerhalb des Vektors enthalten. Da der konstante Teil im Befehl entweder die Basisadresse oder aber ein Indexwert sein kann (oder irgend etwas anderes, was nur dem Programmierer bekannt ist), wird dieser Wert Displacement genannt. Er ist der Unterschied oder das Displacement zwischen der durch das Register bestimmten Adresse und der aktuellen Stelle im Speicher, die adressiert werden soll.

Basis + Index + Displacement

Ein Programm kann auch eine berechnete Adresse mit einem berechneten Index kombinieren. Wie wir in Abbildung 3.2 sehen, kann man Daten auch mit zwei verschiedenen Adressregistern bestimmen. Ein Befehl kann also jedes der beiden Basisregister (BX oder BP) mit jedem der beiden Indexregister (SI oder DI) kombinieren und so die effektive Adresse errechnen. Zusätzlich kann das Programm auch noch ein Displacement liefern, das auf den Wert, der durch die beiden Register bestimmt ist, addiert werden soll. Diese Adressierungsmethode stellt ein Maximum an Flexibilität dar und gestattet es dem Programmierer, sowohl die Basisadresse als auch den Indexwert erst zur Ablaufzeit des Programms zu bestimmen. Diese Fähigkeit wird zwar nicht immer benötigt, aber wenn sie einmal nötig sein sollte, ist sie vorhanden. Unser Beispiel mit dem Notenbuch zeigt sehr schön, wie in einem Programm sowohl die Basisadresse als auch der Vektorindex erst während der Ablaufzeit festgelegt werden können. Um die Noten für den I-ten Schüler für die J-te Schulaufgabe zu bestimmen, müssen wir die Basisadresse mit dem J-ten Vektor berechnen und dann den Index für das I-te Element bestimmen.

Abbildung 3.3 faßt noch einmal die verschiedenen Adressierungsarten zusammen, die auf dem 8088 möglich sind. Es gibt dabei 8 Variationen. Ein Befehl kann jedes der 4 Adressregister zusammen mit einem Displacement verwenden. Ein Befehl kann aber auch eine Kombination aus einem Basisregister und einem Indexregister zusammen mit einem Displacement verwenden. Die mit R/M überschriebene Spalte werden wir später erklären.

R/M	Operandenadresse
000	[BX + SI + DISPLACEMENT]
001	[BX + DI + DISPLACEMENT]
010	[BP + SI + DISPLACEMENT]
011	[BP + DI + DISPLACEMENT]
100	[SI + DISPLACEMENT]
101	[DI + DISPLACEMENT]
110	[BP + DISPLACEMENT]
111	[BX + DISPLACEMENT]

Abbildung 3.3 Adressierungsarten des 8088

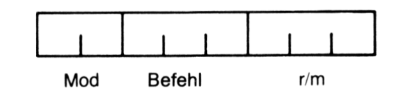
Im Befehlssatz des 8088 wurde die Verwendung des Displacementfeldes für Adressierungszwecke optimiert, um die Anzahl der vom Programm benötigten Bytes zu minimieren. Es kann bei einem Befehl z.B. das Displacementfeld völlig wegfallen, was einem Displacement von 0 entspricht. Ist das Displacement innerhalb des Bereichs von -128 bis 127, so genügt ein einzelnes Byte. Benötigt ein Befehl dagegen den vollen 16-Bit Adressraum, dann umfaßt das Displacementfeld zwei Bytes. Solchermaßen kann das Displacementfeld 0,1, oder 2 Bytes lang sein, je nachdem wie es benötigt wird. Ist das Displacement nur 1 Byte lang, so wird die enthaltene Binärzahl vor der Adressaddition vorzeichenerweitert. Vorzeichenerwei-

tert bedeutet, daß der Prozessor das höchstwertige Bit des Displacementfelds jeweils in die Bits 8-16 überträgt, bevor die Addition durchgeführt wird. Dies erlaubt die Darstellung von auch negativen Zahlen mit nur einem einzigen Displacementbyte. Das schönste an der ganzen Sache ist, daß der Assembler selbst die korrekte Länge ermittelt und die jeweils kürzeste Anweisung in den Befehlscode einsetzt.

Obwohl der 8088 über so viele verschiedene Adressierungsmöglichkeiten verfügt, ist der Befehlssatz so ausgelegt, daß innerhalb eines Befehls nur jeweils eine Speicheradresse angegeben werden kann. Eine Additionsanweiusng mit 2 Operanden muß also ein Register auf eine Speicherstelle bzw. ein Register auf ein anderes Register addieren. Eine einzelne Anweisung kann nie 2 Speicherstellen zugleich ansprechen. Dies bedeutet, daß jeder Befehl nur eine Speicheradresse beinhalten darf.

MOD-R/M-Byte

Wie werden nun alle diese Informationen in die Maschinensprache des 8088 übertragen? Der 8088 verwendet das MOD-R/M-Byte für fast alle Arten der Adressierung eines Operanden. Abbildung 3.4 zeigt uns das Format dieses Bytes. Das MOD-R/M-Byte folgt dem Byte mit dem Befehlscode und erläutert den Speicheroperanden des Befehls. Das MOD-R/M-Byte kann auch ein Register an Stelle einer Speicherstelle benennen. Damit ist es uns nun möglich, alle Adressierungsarten darzustellen.



Mod	Displacement
00	DISP = 0, kein Displacement
01	- 128<DISP<127, Displacement 1 Byte
10	- 32768<DISP<32767, Displacement 2 Bytes
11	r/m ist Registerangabe

Abbildung 3.4 Mod-r/m-Byte

Die ersten beiden Bits des MOD-R/M-Bytes bezeichnen den verwendeten Adressierungsmodus. Die beiden Bits bestimmen die Anzahl der Displacement-Bytes, die dem MOD-R/M-Byte folgen, also entweder 0,1, oder 2. Die letzten drei Bits des MOD-R/M-Bytes bestimmen die Art der Adressierung des Operanden, also die acht verschiedenen Kombinationsmöglichkeiten aus Basis- und Indexregistern, die wir zum Adressieren verwenden. Wir bezeichnen diese drei Bits auch als R/M-, oder Register/Modifier-Feld. Sie bilden die R/M-Spalte in Abbildung 3.3, die uns auch die möglichen Kombinationen der verschiedenen Adressierungsarten zeigt. Die Bedeutung der restlichen drei Bits in der Mitte des MOD-R/M-Bytes hängt vom jeweiligen Befehl ab. Für Befehle mit zwei Operanden, wie z.B. ADD, benennt dieser Bereich das Register, das den zweiten Operanden enthält. Für Befehle mit nur

einem Operanden, wie z.B. INC sind die drei Bits gewöhnlich ein Teil des Befehlscodes. Der 8088 weiß also nicht, daß es sich um eine INC-Anweisung handelt, bevor er nicht die mittleren drei Bits des MOD-R/M-Bytes dekodiert hat.

Es gibt eine ganze Menge von Sonderfällen, die auch durch das MOD-R/M-Byte behandelt werden. Wird beispielsweise in einem Befehl ein Register anstelle eines Speicherbereichs angesprochen, so wird der MOD-Bereich auf 11B gesetzt, um dem 8088 mitzuteilen, daß das R-M-Feld einen Registerwert anstelle einer Speicheradresse enthält. Und schließlich haben Sie sicher festgestellt, daß es keine Möglichkeit gibt, Direktadressierung über das MOD-R/M-Byte darzustellen. Der 8088 behandelt ganz einfach den Fall eines 0-Displacement (BP + Displacement) als direkte Adressierung. In diesem Fall ist das Displacementfeld zwei Bytes lang und kein Register in die Adressberechnung miteinbezogen. Wegen dieses Sonderfalls benötigt der Zugriff auf die Speicherstelle, die durch den Inhalt von BP adressiert wird, ein Displacement von einem Byte mit dem Wert 0. Die gleiche Situation benötigt bei Benutzung der Register BX, SI, oder DI allerdings kein Displacement. Im nächsten Abschnitt werden wir noch einen weiteren Unterschied der Adressierung mit den Registern BP und BX besprechen.

Physikalische Adressierung

Alles, was wir bisher über die Adressierung gehört haben, hat sich mit dem befaßt, was wir als Adressoffset bezeichnen. Dieser Offset ist ein 16-Bit-Wert. Der 8088 segmentiert den Speicher, so daß er in der Lage ist, mehr als jeweils 64K Bytes zu adressieren. Im folgenden werden wir uns genauer ansehen, wie der 8088 diese Segmentierung durchführt.

Da der 8088 eine Wortlänge von 16 Bits aufweist, ist es nur natürlich, daß er Adressen erzeugt, die 16 Bits lang sind. Dies erlaubt die direkte Adressierung von 2^{16} Punkten, oder 65,536 Bytes. Allerdings reichen diese 64K Speicher für manche Programme nicht aus. Deshalb versah Intel den 8088 mit der Möglichkeit, 2^{20} Bytes, oder ein Megabyte Speicher zu adressieren.

Um diese 20-Bit-Adresse zu erhalten, müssen den bestehenden 16 Bits Offset noch weitere vier Bits hinzugefügt werden. Diese zusätzlichen vier Adressbits kommen aus den sogenannten Segmentregistern. Die Segmentregister ihrerseits sind allerdings wiederum nur 16 Bits breit. Der 8088 kombiniert nun den 16-Bit-Adressoffset und das 16-Bit-Segmentregister wie in Abbildung 3.5 dargestellt. Zu diesem Zweck erweitert der Prozessor das Segmentregister um vier Nullbits, und erreicht so einen vollen 20-Bit-Wert. Dann wird auf den erweiterten Segmentwert der Adressoffset, den wir bereits durch die Adressberechnung gewonnen haben, aufaddiert. Das Ergebnis ist ein 20-Bit-Zeiger auf die gewünschte Speicherstelle.

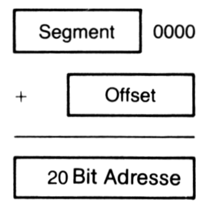


Abbildung 3.5 Segment-Offset Adressberechnung

Jeder Befehl, der sich auf eine Speicheradresse beziehen kann, ist in der Lage, eine 16-Bit-Adresse zu erzeugen. Der Prozessor verwendet nun diesen Offset zum Adressieren innerhalb des gewünschten Segments. Abbildung 3.6 zeigt diese Art der Vorgehensweise.

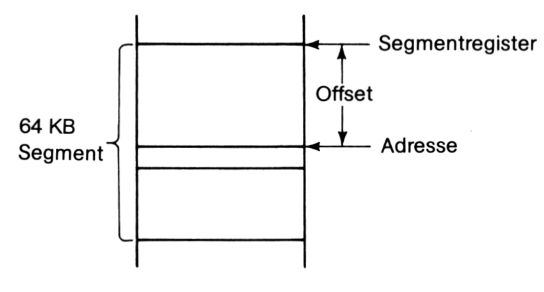


Abbildung 3.6 Segmentierung

Die vier niedrigwertigsten Bits der Anfangsadresse eines Segments stehen grundsätzlich auf Null. Jede sechzehnte Speicherstelle hat eine Adresse mit dieser Eigenschaft. Wenn Sie die Daten für Ihr Programm planen, dann beachten Sie, daß ein Segment immer an diesen 16-Byte-Grenzen beginnen muß. Diese Grenzen werden auch Paragraphgrenzen genannt.

Segmentregister

Der 8088 verfügt über 4 Segmentregister: CS, DS, SS, und ES für Code, Daten, Stack, und Extrasegment. Jedes der Register hat eine Standardverwendung, doch können sie auch entsprechend den Erfordernissen des jeweiligen Programms verwendet werden.

Der 8088 benutzt das Code-Segmentregister, um das Segment zu identifizieren, das den jeweils aktuellen Programmteil enthält. Dieses CS-Register bildet zusammen mit dem Befehlszähler einen Pointer auf die gerade auszuführende Anweisung. Jedes Auslesen eines Befehls geschieht an der Stelle, die durch das Registerpaar CS:IP bezeichnet ist.

Die Kombination eines Segmentregisters mit einem Offsetregister zum Erzeugen einer physikalischen Adresse schreiben wir als Segment:Offset — beispielsweise CS:IP. Dabei steht der Segmentwert vor dem Doppelpunkt, der Offsetwert folgt ihm. Diese Notation wird sowohl für Register als auch für absolute Werte verwendet. Wir können also Adressen schreiben wie CS:100, DS:BX, 570:100, oder 630:DI.

Der Prozessor benützt das Daten-Segmentregister für den normalen Zugriff auf Daten. Die Adressierungsarten für Operanden, die wir bereits besprochen haben, erzeugen immer einen 16-Bit-Offset. In fast allen Fällen kombiniert der Prozessor diesen Offset mit dem Inhalt des DS-Registers, um die Adresse der aktuellen Speicherstelle zu erhalten.

Das Stack-Register bezeichnet den aktuellen Systemstack. Die Befehle PUSH, POP, CALL, und RET greifen grundsätzlich auf Daten zu, die durch das Registerpaar SS:SP definiert sind. Das SP-Register ist in diesem Falle der Stack-Pointer und dient zur Aufnahme des Adressoffsets für das Stack-Segment. Auf das Stack-Segment greifen wir standardmäßig auch dann zu, wenn wir über das BP-Register adressieren. Dies erlaubt es unserem Programm, auf Daten im Stack zuzugreifen und dabei das BP-Register als Pointer zu benützen. Im nächsten Abschnitt werden wir uns mit Stackoperationen befassen, die zeigen, wie wir über das BP-Register Daten im Stack auf einfache Weise ansprechen können.

Und schließlich benützt der 8088 noch das Extra-Segmentregister für Zugriffe auf Daten, wenn wir mehr als ein Segment verwenden. Eine allgemeine Anwendung hierfür ist das Kopieren von Daten aus einem Speicherbereich in einen anderen. Befinden sich diese Daten dabei nicht innerhalb des gleichen 64K-Adressbereichs, ist es unmöglich, die Daten unter Verwendung nur eines einzigen Segmentregisters zu kopieren. Mit Hilfe des Extra-Segmentregisters kann ein Programm nun gleichzeitig sowohl Quell- wie auch Zielsegment ansprechen, wie wir in Abbildung 3.7 sehen. Dabei beinhaltet das DS-Register die Adresse des Segments mit den Ursprungsdaten, und das ES-Register zeigt auf das Zielsegment. Im Befehlssatz des 8088 gibt es einige spezielle Stringanweisungen, die automatisch das DS- und ES-Register für Quell- und Zielsegment verwenden. Im nächsten Kapitel werden wir diese Anweisungen besprechen.

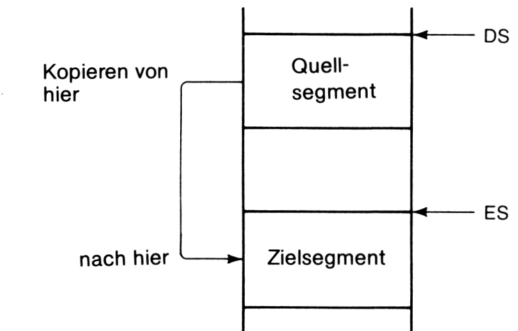


Abbildung 3.7 Datenübertragung von Segment zu Segment

Überschreiben von Segmentregistern

Jedes Segmentregister hat seine, wie bereits gezeigt, normale Verwendung, doch kann es manchmal durchaus angebracht sein, Daten, die nicht im Datensegment liegen, anzusprechen. Ein Beispiel dafür wäre ein kleiner Datenbereich innerhalb eines Programms. Im allgemeinen bearbeitet ein Programm Daten innerhalb eines Speicherbereichs, der über das DS-Register angesprochen wird. Es kann aber durchaus vorkommen, daß ein Programm eine lokale Variable, eine Speicherstelle innerhalb des Programms selbst, ansprechen muß. Zu diesem Zweck muß das Programm die normale Verwendung des Segmentregisters überschreiben. Abbildung 3.8 zeigt diesen Vorgang.

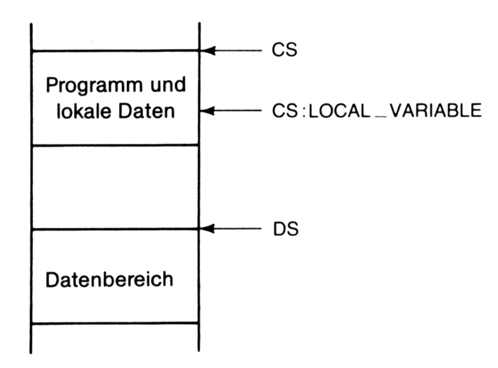


Abbildung 3.8 Verwenden des CS-Registers zum Ansprechen von Daten

Anstatt nun den Inhalt des DS-Registers so zu verändern, daß er auf das Programmsegment zeigt, modifiziert der Befehl die Datenadresse, um damit anzuzeigen, daß die Variable sich innerhalb des Code-Segments befindet.

```
INC     CS:LOCAL_VARIABLE
```

Die Angabe CS: ist für den Assembler der Hinweis, daß sich die Daten im Code-Segment befinden. In der Maschinensprache drückt sich dies so aus, daß der Maschinenbefehl mit einem 1-Byte-Präfix versehen wird. Der 8088 versteht dieses Präfix und ändert daraufhin die normale Methode der Adressberechnung. Der Prozessor verwendet nun das CS-Register zur Bestimmung der physikalischen Adresse anstelle des sonst üblichen DS-Registers. Ein einziges Präfixbyte ist alles, was wir dazu benötigen, da der 8088 ja sowieso jeweils nur einen Speicheroperanden in einem Befehl ansprechen kann.

Ein Befehl kann jedes der vier Segmentregister zur Adressierung von Daten verwenden. Standardmäßig wird hierzu das DS-Register verwendet; das bedeutet, daß das DS-Register verwendet wird, so lange der Befehl kein anderes Segmentregister anspricht. Erinnern wir uns an dieser Stelle, daß die Verwendung des BP-Registers für die Adressberechnung das Stack-Segmentregister zum Standardregister macht. Ein Befehl vermag jedes der drei anderen Register zu ersetzen, soweit es als Teil des Adressausdrucks angegeben wird. Allerdings gibt es auch einige Befehle, in denen das Segmentregister nicht verändert werden darf. Dies sind z.B. die Stringanweisungen. Ein Stringbefehl definiert implizit seine Verwendung der Segmentregister, weshalb sie nicht überschrieben werden dürfen. In Kapitel 4 werden wir die Stringanweisungen und ihre besondere Registerbelegung besprechen.

SEGMENT-Anweisung

Der Assembler unterstützt uns beim Problem der Segmentadressierung. Im Assemblerprogramm muß mit Befehlen angegeben werden, aus welchen Segmenten ein Programm besteht. Über diese Anweisungen teilen wir dem Assembler auch mit, welches Segmentregister zu welchem Segment gehört. Der Assembler kann dann,

falls es benötigt wird, dem jeweiligen Maschinencode das entsprechende Präfix voransetzen. Verwenden wir in unserem Programm eine Variable, die nicht über das DS-Register adressiert werden kann, die aber über ein anderes Segmentregister adressierbar ist, dann generiert der Assembler automatisch das passende Präfix. Dies gestattet uns, auf einfache Weise zu programmieren und dem Assembler die Arbeit zu überlassen, die jeweils passende Adressierungsmethode herauszufinden.

Die Segmentanweisung erlaubt es dem Assembler, festzustellen, welche Segmente gerade aktiv sind, und außerdem mögliche Fehler festzustellen. So könnte z.B. in einem Programm eine Variable angesprochen werden, die nicht zur Verfügung steht, da gerade kein Segmentregister auf das entsprechende Segment zeigt. Der Assembler wird dies als Fehler erkennen. Der Fehler taucht auf, da das Programm die Adressierbarkeit der gewünschten Daten noch nicht hergestellt hat. Obwohl dies manchmal frustrierend sein kann, ist es aber sicher besser, einen Fehler bereits während der Assemblierung festzustellen, als ihn erst später bei der aktuellen Ausführung eines Programms zu bemerken.

```
The IBM Personal Computer Assembler 12-11-82          PAGE    1-1
Figure 3.9  Segments

1                                       PAGE    ,132
2                                       TITLE   Figure 3.9  Segments
3
4        0000                   DATA    SEGMENT
5        0000  01               VAR1    DB      1           ; Variable in Data segment
6        0001                   DATA    ENDS
7
8        0000                   BUFFER  SEGMENT
9        0000  02               VAR2    DB      2           ; Variable in Buffer segment
10       0001                   BUFFER  ENDS
11
12       0000                   CODE    SEGMENT
13       0000  03               VAR3    DB      3           ; Variable in Code segment
14
15                                      ASSUME  CS:CODE, DS:DATA, ES:BUFFER
16
17       0001  FE 06 0000 R             INC     VAR1        ; Variable from Data
18       0005  26: FE 06 0000 R         INC     VAR2        ; Variable from Buffer
19       000A  2E: FE 06 0000 R         INC     VAR3        ; Variable from Code
20
21       000F                   CODE    ENDS
22                                      END
```

Abbildung 3.9 Segmente

Die Segmentanweisung definiert alle Segmente. Außerdem gibt diese Anweisung jedem Segment einen Namen. Das Assemblerprogramm in Abbildung 3.9 zeigt die Definition verschiedener Segmente. Ein Programm darf dabei jeden als Variablennamen gültigen Namen auch als Segmentnamen verwenden. Die Segmentanweisung teilt dem Assembler mit, daß sich alle darauf folgenden Anweisungen und Datenbereiche zur Ausführungszeit des Programms innerhalb desselben Segments befinden werden. Die Anweisung ENDS definiert dagegen das Ende eines Segments. Auch diese Anweisung trägt den Namen des Segments. Außerdem muß für jede Segmentanweisung eine entsprechende ENDS Anweisung gegeben werden. Wenn dem nicht so ist, werden wir den Assembler verwirren, und das Ergebnis wird eine Fehlermeldung sein.

ASSUME-Anweisung

Nachdem wir nun die Segmente definiert haben, muß der Assembler auch wissen, welche Register zur Ausführungszeit des Programms mit diesen Segmenten in Verbindung gebracht werden sollen. Obwohl unser Beispiel im Abbildung 3.9 nur drei Segmente hat, können lange Programme wesentlich mehr Segmente aufweisen. Mit den zur Verfügung stehenden 4 Registern allerdings kann ein großes Programm jeweils nur einen Teil der maximal verfügbaren Segmente gleichzeitig ansprechen. Der Assembler muß deshalb wissen, welche Segmente während der Ausführung adressiert werden. Dies geschieht mittels der ASSUME-Anweisung. Diese Anweisung teilt dem Assembler die Zuordnung der einzelnen Segmentregister zu den jeweiligen Segmenten mit. Der Programmierer muß mit dieser Anweisung selbst die entsprechenden Zuordnungen treffen.

Abbildung 3.9 zeigt diese Vorgehensweise. Das Beispiel umfaßt dabei drei Segmente: DATA, BUFFER und CODE. Die Namen, die wir für diese Segmente wählten, sind beliebig. Unser Programmierer wählte die Namen, für den Assembler dagegen sind sie bedeutungslos. So können wir z.B. ein Segment als CODE bezeichnen und es dann nur für Daten verwenden, oder entgegengesetzt. Das beste ist natürlich, die Segmente so zu benennen, daß ihre Namen innerhalb des Programms sinnvoll erscheinen. In unserem Beispiel haben die Segmente DATA und BUFFER jeweils nur einen einzigen Datenbereich. Das Segment CODE umfaßt einen Datenbereich und außerdem noch drei Befehle. Es ist unwahrscheinlich, daß ein Programm für jeweils einen einzelnen Speicherplatz ein eigenes Segment verwendet, aber für unsere Zwecke genügt dieses Beispiel. Ein Programm benötigt dagegen eine ganze Reihe von Segmenten und Definitionen, wenn es Daten ansprechen soll, die sich über den gesamten vom 8088 adressierbaren Raum ausdehnen. Bei einem Gerätesteuerprogramm für den IBM PC kann es z.B. möglich sein, daß Daten im Systembereich angesprochen werden, ein Interrupt-Vektor im unteren Speicherbereich gesetzt wird, und das Programm selbst an einer ganz anderen Stelle abläuft. Jeder dieser einzelnen Bereiche ist ein eigenes Segment und muß entsprechend im Programm definiert sein.

Die ASSUME-Anweisung in Abbildung 3.9 teilt dem Assembler mit, Maschinencode mit der folgenden Registerzuordnung zu erstellen: Das CS-Register zeigt auf die Anfangsadresse des Codesegments, das DS-Register zeigt auf den Datenbereich, und das ES-Register zeigt auf ein Segment mit dem Namen BUFFER. Die ASSUME-Anweisung teilt dem Assembler mit, er soll eine bestimmte Annahme machen. Der Assembler generiert nämlich Maschinencode in der Annahme, daß die Zuordnung des Segmentregisters genau so sei wie sie mit dieser Anweisung getroffen wird. Die in der ASSUME-Anweisung gemachten Registerzuordnungen bleiben solange gültig, bis wir mit einer weiteren ASSUME-Anweisung eine neue Zuordnung treffen. Der Assembler behandelt ASSUME-Anweisungen also in sequentieller Reihenfolge selbst über Programmschleifen und Sprünge hinweg. Die ASSUME-Anweisung bleibt solange gültig, bis der Assembler beim sequentiellen Durchlaufen unseres Quellcodes die nächste ASSUME-Anweisung erreicht. Halten wir aber fest, daß die ASSUME-Anweisung nicht notwendigerweise alle Segmentregister festlegen muß. Unser Beispiel definiert z.B. das SS-Register nicht. Und in der Tat kann es durchaus

möglich sein, daß während der Ablaufzeit eines Programms die Inhalte bestimmter Register unbekannt sind. Für diese Fälle sollte als Segmentname in der ASSUME-Anweisung der Name NOTHING angegeben werden. So teilt z.B. die Anweisung

```
ASSUME    ES:NOTHING
```

dem Assembler mit, daß das Programm zur Ausführungszeit nicht weiß, auf welches Segment das ES-Register zeigt. Da das Register also unbekannt ist, sollte es der Assembler auch in keiner seiner Adressberechnungen verwenden.

Wichtig ist, daß die ASSUME-Anweisung keinerlei Maschinencode erzeugt. Sie ist lediglich eine Anweisung für den Assembler, anzunehmen, die Segmentregister würden entsprechend der Angabe benützt. Es ist nun Aufgabe des Programmierers, dafür zu sorgen, daß die Segmentregister den richtigen Wert enthalten. In ähnlicher Weise kann der Assembler auch nicht überprüfen, ob die in der ASSUME-Anweisung gemachten Angaben zur Ausführungszeit des Programms mit den aktuellen Registerinhalten übereinstimmen. Da es innerhalb eines Programms viele Wege gibt, die zu einer bestimmten ASSUME-Anweisung führen können, liegt die Korrektheit dieser Anweisung ganz in der Verantwortung des Programmierers. In unserem Beispiel nimmt das Programm an, daß die Segmentregister vor Ausführung korrekt gesetzt sind. Sind die Register nicht richtig gesetzt, wird auch das Programm fehlerhaft arbeiten, obwohl die Assemblierung korrekt war.

Der erste Befehl erhöht den Wert der Variablen VAR1, die sich innerhalb des Datensegments befindet. Entsprechend der ASSUME-Anweisung nimmt der Assembler hier an, daß das DS-Register auf das Segment DATA zeigt. Da das DS-Register außerdem das Standardregister für die Adressierung von Datenbereichen ist, generiert der Assembler auch kein Präfix für diesen Befehl. Die daraus entstehende vier Byte lange Maschinenanweisung enthält also auch kein Segmentpräfix.

Der zweite Befehl spricht die Variable VAR2 an, die wir als im Segment BUFFER befindlich definiert haben. Unser Programm hat dem Assembler bereits mitgeteilt, daß das Extrasegmentregister auf das Segment BUFFER zeigt. Der Assembler generiert also wiederum eine vier Byte lange Maschinensprachenanweisung, um den Wert der Variablen VAR2 zu erhöhen, doch er setzt diesem Befehl ein 1-Byte-Präfix voran, das die standardmäßige Nutzung des DS-Registers verhindert. Das Präfixbyte 26H teilt dem Prozessor also mit, er solle zum Erzeugen der 20-Bit-Speicheradresse das ES-Register verwenden. In der Assemblerliste erkennen wir dieses Präfix an einem Strichpunkt im Maschinencode.

Der dritte Befehl spricht die Variable VAR3 im Segment mit dem Namen CODE an. Die ASSUME-Anweisung hat dieses Segment bereits mit dem CS-Register in Verbindung gebracht. Der Assembler generiert auch hier wieder automatisch das entsprechende Präfix. In diesem Falle würde das Präfix 21H dem Prozessor dann mitteilen, er solle das CS-Register zur Bestimmung der aktuellen Adresse verwenden.

Auf den ersten Blick wird die ASSUME-Anweisung unwichtig erscheinen. Es ist ganz normal, die ASSUME-Anweisung zu vergessen, wenn man das erste Mal ein Programm schreibt. Der Assembler wird dann eine ganze Menge von Fehlermel-

dungen erzeugen, um Sie auf dieses Versäumnis hinzuweisen. Doch im großen und ganzen gesehen hilft die ASSUME-Anweisung dem Programmierer, sich auf die Datenstrukturen zu konzentrieren, die er in seinem Programm verwendet. Der Programmierer darf aber nie vergessen, das entsprechende Segmentregister zu besetzen, wenn er Daten in seinem Programm ansprechen will. Für die meisten Fälle wird der Assembler das passende Segmentregister selbst bestimmen und uns so die Arbeit erleichtern.

Die SEGMENT-Anweisung kann innerhalb eines Programms auch dazu verwendet werden, Informationen an andere Programme zu übergeben. Mit der SEGMENT-Anweisung können wir nämlich die Segmentausrichtung im Speicher bestimmen, die Art der Kombination unseres Segments mit einem anderen Segment, und einen Klassennamen. Es gibt zwei Arten der Segmentausrichtung, die für den Programmierer auf dem IBM PC von besonderem Interesse sind. Paragraphenausrichtung (PARA) stellt das Segment an eine Paragraphengrenze — eine Speicheradresse die durch 16 ohne Rest teilbar ist. Dies bedeutet, daß das erste Byte des Segments einen Offset von 0 zum Inhalt des Segmentregisters aufweist. Im Gegensatz dazu stellt die BYTE-Ausrichtung ein Segment an eine beliebige Stelle im Speicher. In diesem Falle zeigt das Segmentregister nicht unbedingt auf das erste Byte des Segments. Ein Programm, das die erste Speicherstelle dieses Segments ansprechen soll, benötigt also dafür einen Adressoffset, der ungleich 0 ist.

Die Angabe eines Kombinationstyps ermöglicht es außerdem, Segmente auf verschiedene Weise zusammenzubinden. Dies ist für modulares Programmieren besonders nützlich. Die Anweisung PUBLIC bewirkt, daß alle Segmente mit dem gleichen Namen zu einem einzigen großen Segment zusammengebunden werden. So können z.B. alle CODE-Segmente zusammengebunden werden, wie etwa ein Hauptprogramm mit den verschiedensten Unterprogrammen. Eine weitere sinnvolle Kombinationsangabe ist AT, die, zusammen mit einem Adressausdruck, das Segment an eine absolute Adresse plaziert. Solch eine Deklaration ist notwendig, wenn es sich um festliegende Daten handelt, wie z.B. den Interruptvektor im unteren Speicherbereich.

Eine noch ausführlichere Erklärung der Segmentanweisung finden Sie in der Beschreibung des Macro-Assemblers für den IBM PC. Wir werden einige der SEGMENT-Anweisungen in späteren Beispielen verwenden.

Kontrollregister

Der 8088 verfügt über drei 16-Bit-Register, die hauptsächlich für Kontrollzwecke verwendet werden. Dieses sind der Stackpointer (SP), der Befehlszähler (IP), und das Flagregister. Der Prozessor benützt die beiden Pointerregister zur Speicheradressierung während der Programmausführung. Das Flagregister dagegen enthält die einzelnen Bedingungscodes, die ein Programm verwenden kann, um den Ablauf zu steuern.

Befehlszähler

Der Befehlszähler ist ein 16-Bit-Register, das den Adressoffset des nächsten Befehls enthält. Wie wir bereits im vorausgehenden Abschnitt gesehen haben, benützt der Prozessor das CS-Register zusammen mit dem Befehlsregister, um die 20 Bit lange physikalische Adresse zu ermitteln. Dabei bestimmt das CS-Register das Segment des ablaufenden Programms, während der Befehlszähler den Adressoffset enthält.

Da es zwei Register gibt, die an der Bestimmung der Adresse des nächsten Befehls mitwirken, gibt es auch verschiedene Methoden, den Ablauf des Programms zu verändern. Der ganz normale Weg wird dabei nur während der normalen Ausführung eines Befehls gegangen. Während der Prozessor einen Befehl aus dem Speicher liest und ihn ausführt, wird der Befehlszähler um die Anzahl der Bytes des jeweiligen Befehls erhöht. Das Registerpaar CS:IP zeigt also dann auf den nächsten sequentiell folgenden Befehl.

Wenn ein Programm einen Sprungbefehl ausführt, ändert es nun diesen normalen Befehlsablauf. Einer der möglichen Sprungbefehle verändert nur den Inhalt des Befehlszählers und erzeugt auf diese Weise einen Sprung innerhalb des gerade aktuellen Segments. Diesen Sprung nennen wir intrasegmental oder NEAR JUMP. Als Sprungziel benötigen wir dabei nur den neuen Wert für den Befehlszähler. Das CS-Register ist und bleibt unverändert. Ein NEAR JUMP kann also die Steuerung nur innerhalb des gerade aktuellen Segments übertragen, womit die Sprungweite auf ein maximum von 64K Bytes beschränkt ist. Um einen Programmpunkt zu erreichen, der noch weiter entfernt ist, benötigt unser Programm eine besondere Art von Sprungbefehl.

Die zweite Art von Sprungbefehlen nennen wir intersegmental oder FAR JUMP. Bei diesem Sprungbefehl weist der Prozessor sowohl dem CS-Register als auch dem Befehlszähler neue Werte zu. Dieser Sprung erlaubt es uns, ein neues Programm, das sich irgendwo im Adressraum des 8088 befindet, auszuführen. Allerdings müssen wir für diesen Sprung neue Werte sowohl für das CS-Register als auch für den Befehlszähler vorsehen. Für einen direkten Sprung erfordert dies einen 5-Byte-Befehl — ein Byte für den Befehlscode und jeweils zwei Bytes für das CS-Register und den Befehlszähler.

Stackpointer

Der Stackpointer ist ein 16-Bit-Register, das den Adressoffset zur gerade aktuellen Stackspitze festlegt. Der Prozessor benützt das Stackpointer-Register zusammen mit dem Stack-Segmentregister, um die physikalische Adresse zu ermitteln. Dieses Registerpaar (SS:SP) wird für alle Stackreferenzen einschließlich der Befehle PUSH, POP, CALL, und RETURN verwendet. Das Registerpaar SS:SP zeigt also immer auf die aktuelle Spitze des Stacks. Stellt ein Befehl ein neues Wort in den Stack, so wird das SP-Register um 2 erniedrigt. Wird ein Wort aus dem Stack gelesen, erhöht der Prozessor das SP-Register um 2. Der Stack wächst also in Richtung auf die

niedrigeren Speicheradressen. Alle Stackoperationen verwenden Wortwerte. Ein einzelnes Byte kann also weder in den Stack geschoben noch aus diesem gelesen werden.

Der Prozessor verändert das SP-Register, um über den Zustand des Stacks auf dem laufenden zu bleiben. Das SS-Register dagegen wird von keinem der Stackbefehle verändert. Ein Programm kann das SS-Register unabhängig von jeder PUSH- oder POP-Anweisung verändern, wobei das SS-Register immer das Segment bezeichnet, in dem sich der Stack befindet. Dies bedeutet, daß der Systemstack immer auf 64K Bytes begrenzt ist. Setzt beispielsweise ein Programm das SS-Register auf 1000H und das SP-Register auf 0, dann befindet sich das erste in den Stack geschobene Wort an den Stellen 1000:FFFFH und 1000:FFFEH. Alle weiteren Daten, die in den Stack geschoben werden, werden an immer niedrigeren Adressen abgespeichert, bis bei den Adressen 1000:0001H und 1000:0000H das Ende des Stacks erreicht ist. Schreibt unser Programm an diesem Punkt noch ein weiteres Wort in den Stack, so wird dies an den Adressen 1000:FFFFH und 1000:FFFEH abgelegt, was den Anfang unseres Stacks bedeutet. Da der Stack jetzt umgekippt ist, zerstören wir somit alle vorher in ihm gespeicherten Daten. Im Normalgebrauch erreicht der Stack kaum die Länge von 64K Bytes. Verwendet ein Programm z.B. einen Stack von 512 Byte Länge, so könnte es das Registerpaar SS:SP mit dem Wert 1000:0200H vorbelegen. Die erste Stackposition würde sich dann an der Stelle 1000:01FFH befinden, und die letzte an der Stelle 1000:0000H. Legt unser Programm allerdings mehr als 256 Worte im Stack ab, so läuft der Inhalt des Stackregisters über den Wert 0 zum höchsten darstellbaren Wert, was bedeuten würde, daß die jetzt zu speichernden Informationen am anderen Ende des Stacksegments abgelegt werden. Geschieht dies, legt das Programm Stackdaten in einem Speicherbereich ab, der nicht für den Stack reserviert war, in unserem Fall an der Stelle 1000:FFFFH. Zweierlei schlimme Dinge könnten passieren. Der Stack kann Befehlscode oder Daten überschreiben, die nicht zerstört werden dürfen. Oder aber die Stackdaten gehen in einen Bereich des Adressraums, dem kein physikalischer Speicher entspricht. Beide Fälle können Fehler sein, die während des Programmtests äußerst schwer zu finden sind, so daß sich als Lösung empfiehlt, immer genügend Raum für den Systemstack zu lassen. Für den IBM PC wird unter dem Betriebssystem IBM DOS eine Stackgröße von mindestens 128 Bytes empfohlen. Dies schafft genügend Platz für die Stackbedürfnisse der verschiedenen Routinen des DOS und dürfte außerdem auch den Bedürfnissen eines normalen Programms entsprechen.

Flagregister

Das letzte Kontrollregister ist das 16 Bit umfassende Flagregister. Dieses Register enthält bitweise Informationen anstelle einer 16-Bit-Zahl. Die 16 Bits im Flagregister haben jedes seine eigene Bedeutung für den Prozessor. Einige Bits enthalten dabei den Bedingungscode, den der letzte ausgeführte Befehl hinterlassen hat. Ein Programm kann diese Codes verwenden, um den Programmablauf zu steuern. So können wir beispielsweise die Codes prüfen und in Abhängigkeit davon einen anderen Ausführungsweg wählen. Andere Bits im Flagregister signalisieren den

augenblicklichen Zustand des Prozessors. Zusätzliche besondere Befehle erlauben uns die Kontrolle dieser Bits.

Die beste Art, das Flagregister zu erklären, ist wahrscheinlich die Bits eines nach dem anderen zu beschreiben. Der Aufbau des Flagregisters ist in Abbildung 3.10 dargestellt. Wir stellen fest, daß dabei nicht alle Bits bezeichnet sind. Diese nicht bezeichneten Bits sind reserviert — d.h., zur Zeit sind sie noch bedeutungslos. Sie könnten allerdings in einigen späteren Versionen des Prozessors für spezielle Zwecke verwendet werden. Deshalb sollte sich ein Programm auch niemals auf Werte in diesen reservierten Bits beziehen.

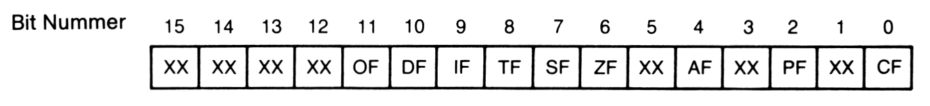


Abbildung 3.10 Flagregister

Alle Flags (=Bits) des niederwertigen Bytes des Flagregisters werden von arithmetischen oder logischen Operationen des Prozessors gesetzt. So setzt z.B. ein Additionsbefehl alle Flags im niederwertigen Byte entsprechend seinem Ergebnis. Mit Ausnahme des Überlaufflags werden dagegen alle Flags im höherwertigen Byte von speziell dafür geschaffenen Befehlen gesetzt oder gelöscht. Die Flags im höherwertigen Byte spiegeln den Zustand des Prozessors 8088 und beeinflussen unter anderem die Art des Programmablaufs. Die Flags im niederwertigen Byte sind Bedingungscodes und können von bedingten Sprungbefehlen abgefragt werden, um dann das Programm entsprechend verzweigen zu lassen.

Vorzeichenflag

Das Vorzeichenflag (Sign Flag — SF) gibt an, ob das Ergebnis der letzten arithmetischen oder logischen Operation positiv oder negativ war. Dabei entspricht das Vorzeichenflag dem Wert des höchstwertigen Bits des letzten Ergebnisses. Falls das letzte Ergebnis negativ war, wird das Vorzeichenflag auf 1 gesetzt. War das letzte Ergebnis positiv oder 0, so wird das Vorzeichenflag auf 0 gesetzt.

Nullflag

Das Nullflag (Zero Flag — ZF) zeigt an, daß das Ergebnis der letzten Operation gleich Null war. Programme benützen dieses Flag beispielsweise um festzustellen, ob zwei Zahlen gleich sind. Zu diesem Zweck werden die beiden Zahlen voneinander subtrahiert. Ist das Ergebnis 0, so waren die beiden Zahlen gleich. Waren die beiden Werte verschieden, so ist das Ergebnis nicht 0.

Parityflag

Das Parityflag (PF) zeigt an, ob die Anzahl der Ziffern im letzten Ergebnis gerade oder ungerade war. Parity ist außerdem eine Methode der Datenprüfung. Das Paritybit ist ein zusätzliches Bit, das den Wert der anderen Bits überprüft. Ein Programm könnte das Parityflag dazu verwenden, um zu entscheiden, ob dieses Extra-Bit gesetzt oder gelöscht werden soll. Der Schreib-/Lesespeicher des IBM PC benützt die Parity beispielsweise, um die darin gespeicherten Werte zu überprüfen. Diese Paritypriifung wird vollkommen von der Systemhardware durchgeführt und beeinflußt in keiner Weise das Parityflag.

Das Parityflag wird auf 1 gesetzt, wenn das Ergebnis eines Befehls aus einer geraden Anzahl von Einsen besteht. Das Parityflag wird dagegen auf 0 gesetzt, wenn die Anzahl der Einsen im Ergebnis ungerade ist. Für normale arithmetische und logische Operationen ist das Parityflag von geringer Bedeutung.

Carryflag

Das Übertragsflag (Carry Flag — CF) ist ein Hilfsmittel des Prozessors für Arithmetik mit höherer Genauigkeit. Der 8088 kann bei arithmetischen Operationen normalerweise ein Maximum von 16 Bits verarbeiten, z.B. bei einer Addition oder bei einer Subtraktion. Es gibt jedoch Fälle, bei denen ein Programm Zahlen verarbeiten muß, die größer als 2^{16} sind. So muß ein Programm beispielsweise, um zwei 32-Bit-Zahlen zu addieren, zuerst den niederwertigen Teil der Zahlen und dann den höherwertigen Teil dieser Zahlen addieren. Abbildung 3.11 zeigt die Addition der beiden 32-Bit-Werte 22223333H und 44445555H.

Zweite Addition	Erste Addition
2222	3333
4444	5555
6666	8888

Abbildung 3.11 32-Bit-Addition

In diesem Beispiel werden als erstes die niederwertigen 16 Bits der beiden Werte addiert, und erzeugen so ein 16-Bit-Ergebnis mit dem Wert 8888H. Im folgenden werden die höherwertigen 16 Bits der beiden Werte addiert, und erzeugen ihrerseits ein Ergebnis von 6666H. Das 32 Bit lange Ergebnis lautet dann 66668888H. Wir benötigen in diesem Beispiel also zwei 16-Bit-Additionen um ein 32-Bit-Ergebnis zu erreichen. Für eine 48 Bit lange Zahl würden wir drei Additionen benötigen, usw. Ein Programm muß folglich jede Zahl, die größer als 16 Bits ist, in 16-Bit Einzelteile aufbrechen.

Unser gezeigtes Beispiel war im übrigen ein sehr einfaches. Das Ergebnis der ersten 16-Bit-Addition hatte nämlich keinen Einfluß auf die zweite 16-Bit-Addition.

Im allgemeinen besteht dagegen grundsätzlich die Möglichkeit, daß ein Übertrag von einer Addition auf die nächste besteht. Bearbeitet der Prozessor eine solche 16-Bit-Zahl, bearbeitet er automatisch auch den Übertrag von einer Position auf die nächste. Addieren wir nun in einem Programm zwei 32-Bit-Zahlen wie in unserem Beispiel zuvor, so müssen wir den Übertrag der ersten 16-Bit-Addition berücksichtigen und ihn bei der zweiten 16-Bit-Addition mitverwenden. Abbildung 3.12 zeigt die Addition der beiden Zahlen 22224444H und 3333EEEEH. Bei diesem Beispiel geht der Übertrag der ersten Additon in die zweite mit ein.

Zweite Addition		Erste Addition
2222		4444
3333		EEEE
1	(Carry aus 1. Addition)	
———		———
5556		1 3332

Abbildung 3.12 32-Bit-Addition mit Übertrag (Carry)

Die erste 16-Bit-Addition, nämlich 4444H mit EEEEH erzeugt als Ergebnis 13332H. Da dieses Ergebnis jedoch 17 Bits lang ist, kann es nicht einem 16-Bit-Register gespeichert werden. Das Carryflag des Statusregisters enthält deshalb dieses zusätzliche Bit arithmetischer Information. Die zweite 16-Bit-Addition addiert dann nicht nur die beiden Zahlen 2222H und 3333H, sondern auch das Carryflag. Es gibt demzufolge zwei Formen der ADD-Anweisung. Die normale ADD-Anweisung addiert zwei 16-Bit-Werte und erzeugt ein 17-Bit-Ergebnis. Der ADC-Befehl (Add with Carry) addiert zwei 16-Bit-Werte und den Übertragswert, um ebenfalls ein 17-Bit-Ergebnis zu erzeugen. In unserem vorhergehenden Beispiel wäre also die erste Addition eine ADD-Anweisung und die zweite Addition eine ADC-Anweisung.

Die Verwendung des Carryflags bei der Arithmetik mit Zahlen von höherer Präzision ist in beiden Beispielen sinnvoll. Das erste Beispiel, die Addition von 3333H mit 5555H, erzeugte einen Übertrag von 0. Wenn nun der ADC-Befehl den Übertrag 0 auf die Werte 2222H und 4444H addiert, erzeugt er ein Ergebnis von 6666H, also den korrekten Wert. In unserem zweiten Beispiel ist das Carryflag gesetzt, da von der niederwertigen Addition ein Übertrag in die höherwertige auftritt.

Für Arithmetik von noch höherer Genauigkeit kann unser Programm das Carryflag wieder und wieder verwenden. Dabei setzt jede 16-Bit-Addition das Carryflag entsprechend ihrem Ergebnis. Das Programm kann dann wiederum auf den nächst höherwertigen Teil der Zahl dieses Carryflag addieren. In jedem Fall enthält das Carryflag dabei das siebzehnte Bit des vorausgehenden Ergebnisses. Und unser Programm muß diesen Wert bei der Addition auf den nächst höherwertigen Teil der Zahl berücksichtigen. Das Carryflag hat noch eine weitere, sinnvollere Verwendung. Auch wenn unser Programm eine Subtraktion ausführt, besteht die Möglichkeit des Übertrags bzw. „Borgens" von einer Zahlenposition auf die nächste. Und das Carryflag zeigt dies auch während der Subtraktion an.

Ein Programm ist dabei in der Lage, Zahlen, die sich über mehrere Speicherwörter erstrecken, ebenso zu subtrahieren wie wir sie vorher addiert haben. Dabei werden die niederwertigen Teile der Zahlen voneinander subtrahiert und bilden ein 16-Bit-Ergebnis. Die Subtraktionsanweisung (SUB) setzt das Carryflag, um den Vorgang des Borgens anzuzeigen. Ähnlich wie bei der Addition verfährt unser Programm dann bei der nächsten 16-Bit-Subtraktion und berücksichtigt dabei diesen Übertrag. Die Anweisung SBB (Subtract with Borrow) führt dabei eine normale Subtraktion aus und zieht zusätzlich den Inhalt des Carryflags vom Ergebnis ab. Ähnlich der Addition können wir auch bei der Subtraktion Zahlen von beliebiger Größe bearbeiten, wenn wir das Carryflag zu Hilfe nehmen.

Der 8088 behandelt das Carryflag wirklich als Zahl. Das bedeutet, wenn es auf Grund einer Subtraktion zu einem Übertrag kommt, setzt der Prozessor das Carryflag auf 1. Unser Programm muß also den Wert 1 vom Ergebnis der nächst höherwertigen Subtraktion abziehen. Gibt es kein „Borgen", so setzt der Prozessor das Carryflag auf 0. Dies bedeutet dann, daß unser Programm vom Ergebnis der nächst höherwertigen Subtraktion nichts abzuziehen hat.

Der Prozessor setzt also das Carryflag als „Borge"-Anzeige für die nächstfolgende Subtraktion. Neben der Tatsache, daß wir auf diese Weise die Rechengenauigkeit des Prozessors erhöhen können, können wir das Carryflag auch verwenden, um die Wertfolge zweier Zahlen festzustellen. Ist nämlich das Carryflag gesetzt, so ist der Subtrahend größer als der Minuend. Ist das Carryflag aber nicht gesetzt, bedeutet dies, daß der abzuziehende Wert kleiner oder gleich dem Wert ist, von dem er abgezogen werden soll. Das heißt, das Carryflag ist eine Anzeige und Entscheidungshilfe bei der Feststellung des relativen Werts zweier Zahlen. Nach der Subtraktion zweier Zahlen teilt uns das Carryflag nämlich mit, welche der beiden Zahlen die größere ist. In einem Programm können wir diese Eigenschaft verwenden, um vorzeichenlose ganze Zahlen zu prüfen, aber auch für Zwecke wie das Sortieren von Zeichenketten. Für vorzeichenbehaftete Zahlen würde unser Programme zusätzliche Informationen benötigen, um die relative Größe dieser Zahlen feststellen zu können. Im nächsten Kapitel werden wir im Abschnitt „bedingte Sprünge" alle Arten besprechen, auf die man Zahlen testen kann.

Hilfs-Carryflag

Sie werden vermutlich niemals in die Verlegenheit kommen das Hilfs-Carryflag (Auxiliary Carry Flag — AUX) zu verwenden, zumindest nicht direkt. Wenn wir uns die bedingten Sprungbefehle des 8088 vornehmen, werden wir sehen, daß es keine Möglichkeit gibt, dieses Flag direkt zu überprüfen. Der Prozessor verfügt über das Hilfs-Carryflag aus einem ganz bestimmten Grunde. Das AUX-Flag erlaubt es ihm nämlich, sogenannte binärkodierte Dezimalarithmetik (BCD) durchzuführen.

BCD-Arithmetik ist völlig verschieden von der Art zu rechnen, wie wir es in Kapitel 2 besprochen haben. BCD-Arithmetik basiert nämlich auf dem Dezimalsystem. Der Prozessor bearbeitet dabei BCD-Zahlen, indem er jeweils Einheiten von vier Bits verwendet, um eine einzelne dezimale Ziffer darzustellen. Jedes Halbbyte kann

dabei die Werte zwischen 0 und 9 darstellen. Die Werte 0AH bis 0FH werden niemals verwendet. Dies bedeutet, daß ein einziges Byte die Werte zwischen 0 und 99 beinhalten kann.

Auf den ersten Blick kann diese Methode, Zahlen darzustellen, platzverschwenderisch erscheinen, denn in jedem Halbbyte werden 6 der 16 möglichen Werte bei der Anwendung von BCD nicht verwendet. Allerding besteht bei vielen Mikroprozessoranwendungen eine direkte Interaktion mit manuellem Input. Wir sind normalerweise gewöhnt, mit Dezimalzahlen zu arbeiten anstelle von Binär- oder Hexadezimalzahlen. Da bei einer bestimmten Anwendung die Ein-/Ausgabeanforderungen durchaus die Anforderungen überwiegen können, die an Speicher- oder Rechenbedarf gestellt werden, kann es günstig sein, Informationen in einer Form aufzubewahren, die leicht für Eingabe oder Ausgabe aufbereitet werden kann. Die BCD-Darstellung kommt diesen Erfordernissen entgegen. BCD-Arithmetik ist außerdem in der Lage, ein Problem zu lösen, das in einigen Binärsystemen auftreten kann. Verwenden wir nämlich Gleitpunktzahlen, kann es durchaus Rundungsprobleme geben, da dieses System der binären Zahlendarstellung nicht direkt unserem Dezimalsystem entspricht. In einigen Fällen können so die Ergebnisse ganz einfacher Rechenvorgänge in der letzten Ziffernstelle falsch sein. Dies wird manchmal auch als das Problem des „fehlenden Pfennigs" bezeichnet, da es hauptsächlich in Rechnungsschreibungsprogrammen auftritt. Das BCD-Zahlensystem stellt in diesem Falle eine Alternative dar.

Der Befehlssatz des 8088 weist keine speziellen Anweisungen für Addition und Subtraktion von BCD-Zahlen auf. Das bedeutet, für BCD-Arithmetik werden die ganz normalen Additions- und Subtraktionsanweisungen verwendet, wie wenn diese Binärzahlen wären. Dabei kann das Ergebnis der Addition von zwei BCD-Zahlen mit dem normalen Arithmetikprozessor eine Zahl sein, die im BCD-Code nicht gültig ist. Um diesen Zustand zu beheben, verfügt der 8088 über verschiedene Anweisungen, die das Ergebnis einer solchen Rechenoperation wieder in den korrekten BCD-Code zurückführen. Um eine solche Umwandlung erfolgreich durchzuführen, muß der Prozessor wissen, ob es z.B. bei einer Addition einen Übertrag vom niederwertigen Halbbyte in das höherwertige Halbbyte gab. Das Beispiel in Abbildung 3.13 zeigt diese Notwendigkeit.

```
  38
+ 29
  —
  61
```

Abbildung 3.13 Dezimale Addition

Die dezimale Addition von 38 und 29 ergibt 67. Die Binäradditon von 38H und 29H ergibt jedoch 61H. Sind die Zahlen 38 und 29 nun BCD-Werte, so muß unser Programm nach der Addition eine Berichtigung vornehmen. Die DAA-Anweisung (Decimal Adjust for Addition) nimmt unser Ergebnis und konvertiert es wieder in das richtige BCD-Format. In unserem Fall sind die Ausgangszahlen im korrekten BCD-

Format — d.h., beide Halbbytes enthalten Werte zwischen 0 und 9. Dagegen ist das Ergebnis falsch. Bei dieser besonderen Art der Addition wird nun das AUX-Flag gesetzt, um anzuzeigen, daß es einen Übertrag vom niederwertigen Halbbyte in das höherwertige Halbbyte gibt. Das Flag teilt der DAA-Anweisung dann mit, daß auf das Ergebnis 6 zu addieren sind, was den korrekten Wert, nämlich 67, ergibt.

Umgekehrt verwendet der Prozessor das AUX-Flag auch für die Korrektur von BCD-Arithmetik nach einer Subtraktion. Dazu dient die DAS-Anweisung (Decimal Adjust for Subtraktion). Es gibt noch zwei weitere Befehle, die auf das AUX-Flag zugreifen. Diese Befehle, AAA (ASCII Adjust for Addition) und AAS (ASCII Adjust for Subtraktion), führen beide die gleiche Art von BCD-Arithmetik aus wie die bereits besprochenen Anweisungen DAA und DAS. In Programmen verwenden wir AAA und AAS für eine Zahlendarstellung, bei der jede Ziffer in einem einzelnen Byte gespeichert ist. Diese Darstellung, die wesentlich mehr Speicherplatz benötigt als die normale BCD-Darstellung, erlaubt uns eine sehr einfache Konvertierung von Zeichendarstellung in Zahlendarstellung und umgekehrt. Im ASCII-Code werden nämlich die Zahlen von 0 bis 9 durch die Werte 30H bis 39H dargestellt. Die Umwandlung von ASCII-Zeichen in Zahlen ist also nur eine einfache Addition oder Subtraktion von 30H zum jeweiligen Zeichenwert. Im folgenden Kapitel werden wir die Verwendung dieser Anweisungen genauer besprechen.

Überlaufflag

Das Überlaufflag (Overflow Flag — OF) ist das einzige Flag im höherwertigen Byte des Flagregisters, das von normalen arithmetischen Operationen gesetzt wird. Alle anderen Flags in diesem Byte des Registers können direkt vom Programmierer beeinflußt werden. Das Überlaufflag ist dabei ein weiteres arithmetisches Flag ebenso wie das Null- und das Carryflag. So wie das Carryflag wichtig ist für die Arithmetik mit Zahlen von höherer Genauigkeit, so ist das Überlaufflag notwendig für das Rechnen mit dem Zweierkomplement.

Arithmetik mit dem Zweierkomplement benutzt nämlich das höchstwertige Bit, um das Vorzeichen der jeweiligen Zahl darzustellen. Die Additionseinheit des Prozessors ist in der Lage, sowohl vorzeichenbehaftete als auch vorzeichenlose Zahlen zu verarbeiten. Normalerweise ergibt die Addition von vorzeichenbehafteten Zahlen ein korrektes Ergebnis. Doch gibt es einige Zahlen beim Rechnen mit dem Zweierkomplement, die, wenn man sie aufeinander addiert, ein falsches Ergebnis erzeugen. Das Beispiel in Abbildung 3.14, die Addition zweier Zahlen im Zweierkomplement, zeigt warum.

Hexadezimal	Dezimales Äquivalent
72H	114
+ 53H	+ 83
0C5H	197

Abbildung 3.14 8-Bit-Addition mit Überlauf

Wenn die Zahlen 72H und 53H vorzeichenlose Zahlen sind, ist das Ergebnis der Addition korrekt. Sind die Zahlen dagegen vorzeichenbehaftete Zahlen im Zweierkomplement, ist das Ergebnis falsch. Das Ergebnis, 0C5H, entspricht —59 in der Zahlendarstellung im Zweierkomplement. Die Addition von zwei positiven Zahlen darf jedoch niemals eine negative Zahl ergeben. Was hier geschehen ist, zeigt, daß das Ergebnis der Addition nicht im 8-Bit-Zweierkomplement-Wertebereich (127 bis —128) dargestellt werden kann. Dieser Effekt wird als Überlauf bezeichnet, da die Addition ganz einfach den Darstellungsbereich der Zweierkomplementzahlen überzogen hat.

Es ist wichtig, festzuhalten, daß Überlaufs- und Carryflag zwei verschiedene Flags sind, und daß sie auch zwei verschiedene Bedeutungen haben. In Abbildung 3.14 gibt es keinen Übertrag, da die vorzeichenlose Addition der beiden Zahlen ein korrektes Resultat erzeugt. Es gibt jedoch einen Überlauf, da die Addition der beiden Zahlen mit Vorzeichen ein ungültiges Ergebnis liefert. Im Verlauf einer einzigen Addition ist es also möglich, sowohl Überlauf als auch Übertrag zu erzeugen, wie uns Abbildung 3.15 zeigt.

Hexadezimal	Zweierkomplement	Vorzeichenlos
8EH	−114	142
0ADH	− 83	173
1 3BH	−197	315

Abbildung 3.15 8-Bit-Addition mit Übertrag (Carry) und Überlauf

Hier werden zwei negative Zahlen addiert. Das Ergebnis, —197, liegt außerhalb des Darstellungsbereichs mit dem Zweierkomplement. Dies ersehen wir an dem 8-Bit Ergebnis, 3BH, das eine positive Zahl wäre. Dieses Beispiel setzt zusätzlich das Carryflag, um anzuzeigen, daß das Ergebnis der vorzeichenlosen Addition eine Zahl ist, die außerhalb des darstellbaren Maximums liegt. Im Fall von 8-Bit-Zahlen wäre diese Grenze nämlich bei 255.

Zusammenfassend können wir sagen, daß die Addition von vorzeichenbehafteten, vorzeichenlosen und auch BCD-Zahlen im Prozessor gleich abläuft. Das Carry-, AUX- und Überlaufflag enthalten dabei Informationen, die es unserem Programm erlauben, den korrekten Wert innerhalb des jeweiligen verwendeten Zahlensystems festzulegen. Das Überlaufflag zeigt an, daß das Ergebnis einer arithmetischen Operation den Bereich der darstellbaren vorzeichenbehafteten Zahlen verlassen hat. Überlauf ist dabei etwas anderes als Übertrag, der anzeigt, daß ein Übertrag über das höchstwertige Bit hinaus aufgetreten ist.

Fallenflag

Das Fallenflag (Trap Flag — TF) hilft uns, Fehler in unserem Programm zu finden. Dieses Flag wird nicht automatisch vom Prozessor gesetzt. Unser Programm kann dieses Flag, das auch als „Trace Flag" oder Einzelschrittflag bezeichnet wird, mit einem speziellen Befehl setzen.

Ist dieses Flag aktiv, tritt nach jedem ausgeführten Befehl eine Unterbrechung ein. Der Effekt ist der gleiche, wie wenn nach jedem ausgeführten Befehl ein externes Gerät eine Unterbrechung anfordern würde. Die Trace-Unterbrechung überträgt die Steuerung an die im Unterbrechungsvektor 4 genannte Speicherstelle. Als Teil des Unterbrechungsprozesses setzt der Prozessor das Fallenflag automatisch zurück. Dies erlaubt es der Trace-Unterbrechungsroutine, abzulaufen, ohne selbst nach jeder ausgeführten Anweisung unterbrochen zu werden. In dem Augenblick, indem die Trace-Unterbrechungsroutine die Steuerung wieder an das Benutzerprogramm übergibt, stellt sie den originalen Inhalt des Benutzer-Flagregisters wieder her, in dem das Trace-Flag immer noch gesetzt ist. Der Prozessor führt dann die nächste Benutzeranweisung aus, und die Fallenunterbrechung tritt wiederum auf. Die Trace-Unterbrechungsroutine übernimmt nach jeder ausgeführten Anweisung des Benutzerprogramms die Kontrolle, solange, bis das Benutzerprogramm das Fallenflag wieder löscht.

Das DOS Debug-Programm benutzt dieses Fallenflag. Eine der Funktionen des Debug-Programms ist die Einzelschrittverarbeitung, die jeweils eine einzelne Anweisung des Benutzerprogramms abarbeitet und dann die Steuerung wieder an den Monitor übergibt. Diese Unterbrechung wird über das Fallenflag erzeugt. Eine genaue Erläuterung des Unterbrechungsprozesses geben wir später im Abschnitt „Unterbrechungsvektor".

Unterbrechungsflag

Das Unterbrechungsflag (Interrupt Flag — IF) steuert externe Unterbrechungen. Während des Ablaufs der Teile eines Benutzerprogramms, in denen es nicht wünschenswert ist, externe Unterbrechungen zuzulassen, kann dieses Flag vom Programm gelöscht werden. Solange das Unterbrechungsflag auf 0 steht, können keine externen Unterbrechungen auftreten. Setzt unser Programm dagegen das Unterbrechungsflag auf 1, können externe Geräte Unterbrechungen erzeugen. Das Benutzerprogramm hat also die Verfügung über das Unterbrechungsflag.

Der IBM PC verfügt über mehrere Methoden, Unterbrechungen zu steuern. Das Unterbrechungsflag des Statusregisters verhindert alle externen Unterbrechungen mit Ausnahme derjenigen, die durch Speicherfehler bedingt sind. Für Fälle, in denen ein Programm nur bestimmte Unterbrechungen ausschalten möchte, ist ein spezielles Unterbrechungsmaskenregister vorgesehen. Über dieses Maskenregister ist es möglich, bestimmte individuelle Unterbrechungen ein- oder auszuschalten. Im Kapitel 8, in dem wir die Hardware des IBM PC beschreiben, werden wir auch dieses Register besprechen.

Richtungsflag

Das letzte Flagbit in unserem Flagregister ist schließlich das Richtungsflag (Direction Flag — DF). Der Befehlssatz des 8088 umfaßt nämlich etliche Stringanweisungen, die mit jeweils einem großen Block von Daten arbeiten. Die Befehle behandeln die Datenblöcke jeweils byte- oder wortweise. Indexregister zeigen in diesem Fall auf die einzelnen Datenblöcke. Nach jeweils einem Verarbeitungszyklus, der ein Byte oder ein Wort umfassen kann, verändert der Prozessor den Inhalt der Indexregister, damit sie auf das jeweils nächste Element im Datenblock zeigen.

Diese Stringoperationen benutzen nun das Richtungsflag, um die Richtung festzulegen, in der die Verarbeitung durch den jeweiligen Datenblock abläuft. Steht das Richtungsflag auf 0, dann erhöhen die Stringanweisungen die jeweiligen Indexregister. Steht das Richtungsflag jedoch auf 1, dann erniedrigen die Stringanweisungen die jeweiligen Indexregister. Über das Richtungsflag ist es also möglich, mit einem einzigen Satz von Stringoperationen in beiden Richtungen zu arbeiten, abhängig jeweils vom Zustand des Richtungsflags. Es kann nämlich unter bestimmten Umständen wünschenswert sein, einen String mit aufsteigenden Adressen zu verschieben, während es in anderen Fällen nötig sein kann, einen String mit absteigenden Adressen zu verschieben.

Wählen wir als Beispiel, daß unser Programm mit Hilfe der Stringoperation einen Datenblock an eine neue Speicherstelle schieben soll. Verschieben wir dabei den Datenblock an eine niedrigere Speicheradresse, wird das Richtungsflag gelöscht, und die Indexregister werden nach jedem Datentransport erhöht. Verschieben wir den Datenblock an eine höhere Speicheradresse, so ist das Richtungsflag gesetzt, und zeigt somit an, das nach jedem Datentransport die Indexregister erniedrigt werden sollen. Für die meisten Transportbefehle ist es übrigens bedeutungslos, ob dieses Richtungsflag gesetzt ist oder nicht. Sollten jedoch Anfangs- und Endadresse eines Datenblocks sich überlappen, und das Richtungsflag ist nicht korrekt gesetzt, so können Teile oder die gesamte Information in diesem Datenblock während des Transportes zerstört werden.

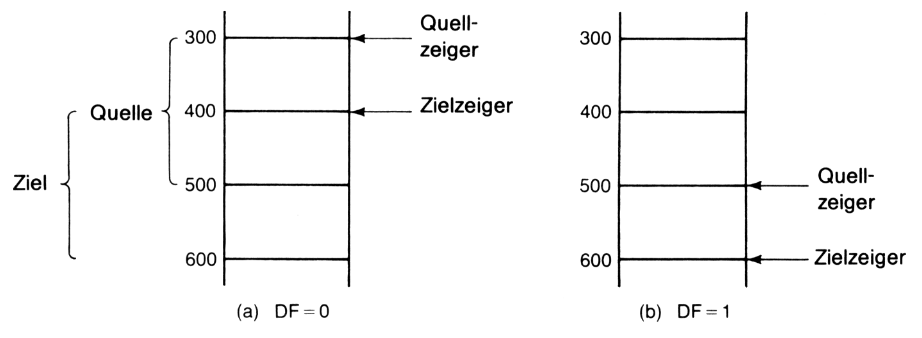


Abbildung 3.16 Richtungsflag

Abbildung 3.16 zeigt ein Beispiel für einen Blocktransport. Der Originaldatenblock hat eine Länge von 200H Bytes, und liegt zwischen den Speicherstellen 300H und 4FFH. Unser Ziel ist es nun, ihn an eine neue Stelle zu transportieren, und zwar

zwischen die Stellen 400H und 5FFH. Quell- und Zielbereich überschneiden sich also.

In Abbildung 3.16 (a) zeigen die Quell- und Zielzeiger auf den Anfang des jeweiligen Datenblocks. Der Quellzeiger enthält also den Wert 300H und der Zielzeiger den Wert 400H. In unserem Beispiel ist das Richtungsflag gelöscht, so daß die Zeiger nach jedem Datentransport erhöht werden. Wie das Bild zeigt, hat nach einem Transport von 100H Bytes der Quellzeiger den Anfang des Zielblocks erreicht. Dieser Bereich des Zieldatenblocks ist jedoch bereits mit neuen Daten gefüllt. Die Übertragung der letzten 100H Bytes wird nun nicht mehr korrekt ablaufen, da wir an Stelle der Originaldaten bereits in den Zielblock übertragene Daten erneut in den Zielblock schreiben.

In Teil (b) unseres Beispiels sind Ziel- und Quellzeiger auf die Enden des jeweiligen Datenblocks gerichtet. Das Richtungsflag ist gelöscht, so daß die Zeiger nach jedem Transport erniedrigt werden. Auf diese Weise werden die Daten im Beispiel korrekt vom Quell- nach dem Zielblock übertragen.

In den Ein-/Ausgaberoutinen des IBM PC wird das Richtungsflag außerdem noch speziell für Bildschirmroutinen verwendet. Diese Routinen verwenden die Stringbefehle des 8088, um Daten innerhalb des Bildschirmpuffers hin- und herzuschieben. Soll der Bildschirminhalt nach oben geschoben werden, werden einfach die Daten an niedrigere Speicheradressen bewegt. Soll der Bildschirm dagegen nach unten verschoben werden, so werden die Daten an höhere Speicheradressen transportiert. In jedem Fall wird dazu ganz einfach das Richtungsflag entsprechend der gewünschten Richtung des Datentransports gesetzt.

Unterbrechungsvektoren

Ein wichtiger Aspekt des 8088 ist seine Unterbrechungsstruktur. Diese Systemkomponente ist fest in den Prozessor eingebaut und gestattet es uns auf effiziente Weise, Unterbrechungen abzuarbeiten.

Erhält der 8088 eine Unterbrechungsanforderung, stellt die Hardware in einem Vorgang, den wir als „interrupt acknowledge cycle" - Unterbrechungsbestätigung bezeichnen, fest,welches Gerät eine besondere Aktion erfordert. Im IBM PC bearbeitet ein eigener Unterbrechungs-Steuerchip, der Intel 8259, externe Unterbrechungsanforderungen. Dieser Unterbrechungscontroller ist darauf programmiert, in Beantwortung auf die Unterbrechungsbestätigung des 8088 eine 1-Byte-Nummer zu liefern. Diese Zahl im Wert zwischen 0 und 255 ist die Unterbrechungsnummer des jeweiligen externen Gerätes, das die Unterbrechung hervorgerufen hat. Für den PC gibt es 8 externe Unterbrechungen, die vom Unterbrechungscontroller behandelt werden und jeweils Werte zwischen 8 und 15 zurückgeben.

Erhält der 8088 nun diese Unterbrechungsnummer zurück, muß er die Steuerung an eine geeignete Unterbrechungsroutine übergeben. Zu diesem Zweck hat der 8088 die ersten 1024 Bytes Speicherbereich für einen Unterbrechungsvektor reserviert. Jeder der 256 möglichen Unterbrechungen ist dabei ein 4-Byte-Spei-

cherbereich zugeteilt. Unterbrechung 0 verfügt über die Bytes 0 bis 3, Unterbrechung 1 verfügt über die Bytes 4 bis 7, usw. Jede dieser 4-Byte-Stellen enthält einen Zeiger auf eine Unterbrechungsroutine für die zugehörige Unterbrechung. Die beiden ersten Bytes enthalten dabei jeweils den Adressoffset und die beiden letzten Bytes die Segmentadresse. Die DD-Anweisung erzeugt dazu die korrekten Adresswerte für Segment und Offset.

Analog zu einem Unterprogrammaufruf muß eine Unterbrechung auch die Rückkehradresse im Stack speichern. Da sich außerdem die Unterbrechungsroutine irgendwo im Adressraum des Mikroprozessors befinden kann, muß der Prozessor die Unterbrechung als einen FAR CALL ausführen. D.h., bevor der Prozessor die Steuerung an die Unterbrechungsroutine übergibt, muß er sowohl Segment als auch Adressoffset des laufenden Programms sichern. Außerdem muß sich der Prozessor nach Rückkehr aus einer Unterbrechungsroutine in genau dem Zustand befinden, indem er sich befand, als die Unterbrechung eintrat. Zu diesem Zweck sichert der 8088 auch das Flagregister in den Stack. Die einzelnen Unterbrechungsroutinen brauchen sich also darum nicht mehr zu kümmern. Das Sichern des Flagregisters sichert aber automatisch auch den aktuellen Status des Unterbrechungszustandsbits. Die Annahme einer externen Unterbrechung löscht das Unterbrechungszustandsbit, so daß die Unterbrechungsroutine selbst von keiner anderen Unterbrechung mehr gestört werden kann. Die Anweisung IRET (Return From Interrupt), die das Flagregister wieder zurückspeichert, schaltet das Unterbrechungssystem automatisch wieder ein, da sie auch das Unterbrechungszustandsbit wieder so herstellt, wie es vor Annahme der Unterbrechung war.

Tritt eine Unterbrechung ein, so speichert der Prozessor das Flagregister in den Stack, gefolgt von CS- und IP-Register. Der 8088 benützt die Unterbrechungsnummer, um einen Zeiger auf die Unterbrechungsroutine zu bilden, und übergibt dann die Steuerung an diese. Jetzt liegt es in der Verantwortung der Unterbrechungsroutine, alle die Register zu sichern, die sie verändern könnte, und sie auch ebenso wiederherzustellen, bevor sie in das unterbrochene Programm zurückkehrt. Der Befehl IRET ist eine spezielle Rückkehranweisung, die nur für die Rückkehr aus Unterbrechungsroutinen verwendet wird. Die Anweisung IRET überträgt die obersten drei Worte aus dem Stack und speichert sie in die Register IP, CS und das Flagregister. Wir werden in späteren Kapiteln noch auf etliche Beispiele eingehen, die diesen Unterbrechungsmechanismus verwenden.

Wir können den Unterbrechungsmechanismus auch direkt ohne äußere Unterbrechungen verwenden. Es gibt nämlich Befehle, die den Prozessor veranlassen, so zu arbeiten wie wenn eine Unterbrechung von außen eingetreten wäre. Diese Unterbrechungen nennen wir Software-Interrupts, da sie zwar durch die Software ausgelöst werden, aber in ihrem gesamten Ablauf einer normalen Unterbrechung gleichen. Der Prozessor speichert also auch hier die drei Steuerregister in den Stack und ermittelt über den vom Programm gelieferten 1-Byte-Wert den passenden Unterbrechungsvektor. Er verwendet dabei den im unteren Speicherbereich vorhandenen Unterbrechungsvektor als Zeiger auf die jeweilige Unterbrechungsroutine.

Software-Interrupts erschließen uns völlig neue Möglichkeiten innerhalb des 8088-Systems. Im Falle eines normalen Unterprogrammaufrufs muß der Programmierer nämlich wissen, wo im Speicher sich das jeweilige Unterprogramm befindet, bevor er es aufruft. Rufen wir unser Unterprogramm aber über einen Software-Interrupt auf, so darf es sich irgendwo innerhalb des erlaubten Adressraums befinden. Das aufrufende Programm hat dabei nicht zu wissen, wo sich das Unterprogramm befindet. Die Nummer des Unterbrechungsvektors ist der einzige Wert, der beim Aufruf bekannt sein muß. Die IBM-Steuerprogramme und das Betriebssystem nutzen dieses Konzept mit großem Vorteil. Softwareunterbrechungen erschließen uns den Zugang zu allen Serviceroutinen des Systems. Das Benutzerprogramm braucht dazu niemals die exakte Adresse zu wissen, die sich im übrigen auch durch verschiedene Neuauflagen der Systemsoftware verändern könnte. Das bedeutet auch, daß diese Serviceroutinen zu jeder Zeit dadurch ersetzt werden können, daß man ganz einfach den 4-Byte-Vektor so verändert, daß er auf die neue Routine zeigt. Die einzelnen Programme, die diese Routine verwenden, müssen dadurch nicht extra verändert werden. In Kapitel 10 werden wir an mehreren Beispielen den Nutzen dieser Vorgehensweise erläutern.

4 Der 8088-Befehlssatz

In diesem Kapitel werden wir den Befehlssatz des Mikroprozessors 8088 besprechen. Im vorangehenden Kapitel befaßten wir uns mit der Registerbelegung und dem Adressmechanismus des Prozessors. In diesem Kapitel werden wir nun erforschen, mit welchen Befehlen wir diese Hilfsmittel vollständig nutzen können.

Dazu teilen wir die Befehle des 8088 in verschiedene Kategorien ein. Dabei bestimmt die Funktion, die ein einzelner Befehl ausführen kann, seine Zugehörigkeit zur entsprechenden Gruppe. Dieses Kapitel ist also keine Liste des Befehlssatzes mit jeweils einer eigenen Seite pro Anweisung. Für diesen Zweck können Sie das Handbuch des Makro-Assemblers sehr gut verwenden. Wir werden hingegen die einzelnen Befehlsgruppen besprechen und versuchen, den bestmöglichen Nutzen aus den einzelnen Befehlen zu ziehen. Dieser Abschnitt sollte Ihnen also eher eine Vorstellung vermitteln, was Sie mit den einzelnen Befehlen anfangen können, als Ihnen ganz genau zu erklären, wie jeder Befehl in allen Einzelheiten abläuft.

Datentransport

In jeder Hinsicht sind die Anweisungen zum Datentransport die am meisten verwendeten innerhalb des Befehlssatzes eines jeden Computers. Der 8088 ist dabei keine Ausnahme. Ein großer Teil aller Probleme in der Datenverarbeitung besteht ganz einfach darin, Informationen von einer Speicherstelle an eine andere zu transportieren. Ein Programm kann dabei die Daten beim Transport auch verändern, doch die Hauptarbeit liegt ganz einfach im Datentransport. Der folgende Abschnitt behandelt die meisten Datentransportbefehle des 8088, und der Abschnitt über die Stringoperationen den restlichen Teil davon.

Transport

Der MOV-Befehl ist der grundlegende Befehl für alle Arten von Datentransport. Er transportiert ein Datenbyte oder ein Datenwort von einer Speicherstelle in ein Register, von einem Register in eine Speicherstelle, oder von Register zu Register. Der MOV-Befehl kann also vom Programmierer ausgewählte Daten in ein Register oder eine Speicherstelle transportieren.

Der MOV-Befehl bildet in der Tat eine ganze Familie von Befehlen in der Maschinensprache des 8088. Ein Verzeichnis mit allen Maschinenbefehlen des 8088 finden Sie im Anhang A. Ein kurzer Blick in diese Liste enthüllt uns, daß es sieben verschiedene Versionen des MOV-Befehls gibt. Mit dem Operationscode MOV haben wir Zugang zu jeder einzelnen dieser Anweisungen. Anhand der von uns mit dem Befehl gelieferten Operanden ermittelt der Assembler dann den korrekten Maschinenbefehl. Dies ist eine der Ursachen, daß der Assembler unbedingt Operanden für die MOV-Anweisung benötigt. Der Assembler muß nämlich in jedem Fall wissen, welchen Operanden wir verwenden — ein Register, eine Byte-Adresse, eine Wort-

adresse, ein Segmentregister oder irgend etwas anderes. Diese Angaben ermöglichen es dem Assembler dann, den korrekten Befehl in Maschinensprache niederzulegen. Im Falle des MOV-Befehles muß der Assembler aus den sieben möglichen Formen durch unsere Angabe die richtige Form bilden.

Abbildung 4.1 zeigt die verschiedenen Möglichkeiten, mit dem 8088 Daten zu transportieren. Jedes Kästchen steht dabei für ein Register oder eine Speicherstelle. Die Pfeile symbolisieren die Wege des Datentransports, die mit den 8088-Befehlen möglich sind. Der Hauptweg führt dabei vom Speicher in die Register und umgekehrt. In einem Register befindliche Daten können nämlich wirkungsvoller bearbeitet werden als Daten im Speicher. Der Prozessor muß dann nicht bei jedem Zugriff auf die Daten auch zugleich einen Speicherzugriff ausführen. Außerdem können wir bei den Befehlen für den 8088 sowieso nur jeweils einen Speicheroperanden angeben. Deshalb ist es z.B. für eine ADD-Anweisung notwendig, daß sich zumindest einer der beiden Operanden in einem Register befindet. Der 8088 verfügt nämlich nicht über die Möglichkeit, den Inhalt einer Speicherstelle auf eine andere in einer einzigen Anweisung zu addieren.

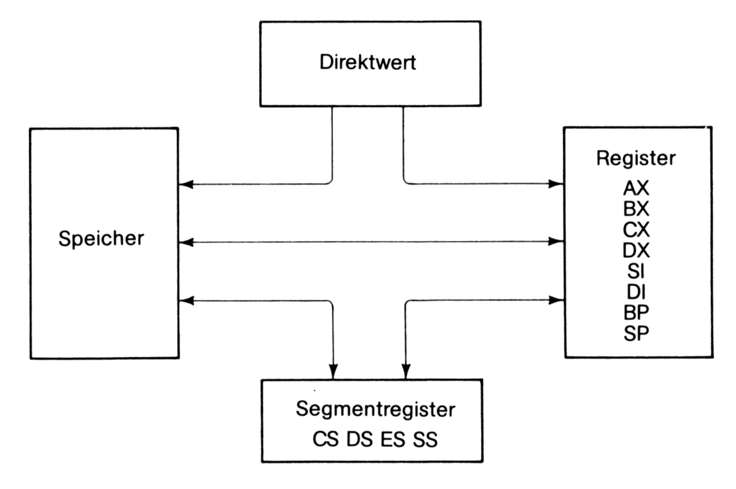


Abbildung 4.1 Ablauf des MOV-Befehls

Ein MOV-Befehl kann außerdem den neuen Datenwert für ein Register direkt im Befehl selbst enthalten. Diese Form des Operanden nennen wir Direktoperand. Die Daten sind nämlich direkt im Befehl verfügbar, eine Adressberechnung ist also nicht mehr nötig. Wir könnten dies auch als eine besondere Art von Adressierung verstehen, bei der sich der Operand an einer Speicherstelle innerhalb eines Befehls befindet, und nicht an einer beliebigen anderen Speicherstelle oder in einem Register. Der 8088 verfügt über solche direkte Befehle sowohl zum Zweck der Datenmanipulation als auch für den Datentransport.

Abbildung 4.1 zeigt, daß ein Befehl Daten direkt in ein Register oder an eine Speicherstelle transportieren kann. Halten wir fest, daß es sinnlos ist, Informationen in einem Befehl abzuspeichern, weshalb der Datenfluß bei direkten Befehlen immer

nur in einer Richtung stattfindet. Schließlich können wir mit dem MOV-Befehl auch noch den Inhalt eines Segmentregisters an eine beliebige Speicherstelle oder in ein anderes Register transportieren. Mit dem Befehl können wir ebenso auch ein Segmentregister aus dem Speicher oder aus einem anderen Register laden. Dagegen gibt es keine Anweisung, über die wir ein Segmentregister mit direkten Daten laden könnten. Es ist im allgemeinen nämlich wenig sinnvoll, ein Segmentregister mit einem direkten Befehl zu laden. Sollte ein Programm dennoch in die Verlegenheit kommen, einen bereits bekannten Wert in ein Segmentregister zu speichern, so muß dieser Wert zuerst in irgendeinem anderen Register oder einer Speicherstelle abgelegt werden. Danach kann in einem zweiten Schritt der Wert in das Segmentregister übertragen werden. In Abbildung 4.2 sehen wir das Vorgehen hierzu.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 4.2 Move instructions

1                                         PAGE    ,132
2                                         TITLE   Figure 4.2 Move instructions
3        0000                    CODE     SEGMENT
4                                         ASSUME  CS:CODE,DS:CODE
5
6        0000                    EXWORD   LABEL   WORD
7        0000                    EXBYTE   LABEL   BYTE
8
9        0000  8B C3                      MOV     AX,BX          ; Register BX --> Register AX
10       0002  8B D8                      MOV     BX,AX          ; Register AX --> Register BX
11
12       0004  8B 0E 0000 R               MOV     CX,EXWORD      ; Memory --> Register
13       0008  89 16 0000 R               MOV     EXWORD,DX      ; Register --> Memory
14
15       000C  8A 2E 0000 R               MOV     CH,EXBYTE      ; Memory --> register (byte)
16       0010  88 36 0000 R               MOV     EXBYTE,DH      ; Register --> Memory (byte)
17
18       0014  BE 03E8                    MOV     SI,1000        ; Immediate --> Register
19       0017  B3 17                      MOV     BL,23          ; Immediate --> Register (byte)
20       0019  C7 06 0000 R 07D0          MOV     EXWORD,2000    ; Immediate --> Memory
21       001F  C6 06 0000 R 2E            MOV     EXBYTE,46      ; Immediate --> Memory (byte)
22
23       0024  A1 0000 R                  MOV     AX,EXWORD      ; Memory --> Accumulator
24       0027  A0 0000 R                  MOV     AL,EXBYTE      ; Memory --> Accumulator (byte)
25       002A  A3 0000 R                  MOV     EXWORD,AX      ; Accumulator --> Memory
26       002D  A2 0000 R                  MOV     EXBYTE,AL      ; Accumulator --> Memory (byte)
27
28       0030  8E 1E 0000 R               MOV     DS,EXWORD      ; Memory --> Segment Register
29       0034  8E D8                      MOV     DS,AX          ; Register --> Segment Register
30       0036  8C 1E 0000 R               MOV     EXWORD,DS      ; Segment Register --> Memory
31       003A  8C C0                      MOV     AX,ES          ; Segment Register --> Register
32
33                               ;-----   Immediate value to segment register
34
35       003C  B8  ---- R                 MOV     AX,CODE        ; Get immediate value
36       003F  8E D8                      MOV     DS,AX          ; Value to segment register
37
38       0041                    CODE     ENDS
39                                        END
```

Abbildung 4.2 MOV-Befehle

Abbildung 4.2 zeigt eine Assemblerliste mit einigen für den MOV-Befehl möglichen Variationen. Halten wir dabei fest, daß der Assemblerbefehl MOV eine ganze Anzahl verschiedener Maschinensprachenbefehle generieren kann.

Werfen wir nun einen Blick auf Abbildung 4.2, die die Syntax der MOV-Anweisung behandelt. Der MOV-Befehl hat zwei Operanden: Ziel und Quelle. Sie müssen auch in dieser Reihenfolge angegeben werden, nämlich zuerst Ziel und dann Quelle. In unserem ersten Beispiel (MOV AX,BX) transportieren wir den Inhalt des BX-Registers in das AX-Register. Der nächste Befehl hat genau den gegenteiligen Effekt. Hier wird nämlich der Inhalt des AX-Register in das BX-Register transportiert. Der

MOV-Befehl verändert dabei niemals den Quelloperanden. So wird durch die Anweisung

```
MOV     AX,BX
```

zwar das AX-Register, das Ziel modifiziert, doch der Inhalt des BX-Registers, der Quelle, wird nicht verändert.

Keiner der MOV-Befehle hat eine Auswirkung auf das Statusflag. Obwohl dies manchmal nicht sehr günstig sein kann, ist es dennoch der beste Weg, mit den Flags umzugehen. Wie wir später noch sehen werden, verfügt der 8088 über genügend Befehle, mit denen beliebige Speicherstellen wirksam überprüft werden können, sodaß wir keinen MOV-Befehl benötigen, um bestimmte Flags zu setzen. Als Beispiel für die Tatsache, daß es sehr unpassend sein könnte, wenn der MOV-Befehl die Flags verändern würde, denken wir nur an Rechenoperationen von höherer Genauigkeit. Wenn ein Programm solche Rechenoperationen ausführt, muß es Teile der Operanden in Register laden, um sie für die Rechenoperationen vorzubereiten. Durch diesen Transport werden keine Flags verändert, was es uns ermöglicht, die Flags dann für das Rechnen mit höherer Genauigkeit zu verwenden.

Wie wir bereits früher erwähnten, gibt es verschiedene Formen der MOV-Anweisung in der Maschinensprache. Der Maschinencode in Abbildung 4.2 zeigt diese verschiedenen Formen. Sollten Sie sich genauer mit der Maschinensprache befassen wollen, dann vergleichen Sie einmal den hier generierten Objektcode mit den Programmlisten im Anhang A. Der Vergleich wird Ihnen die Arbeitsweise der einzelnen Bits in der Maschinensprache erläutern. So können wir z.B. die direkten Datenwerte in den einzelnen Befehlen erkennen. Glücklicherweise benötigen wir zum Programmieren in Assembler nicht unbedingt Kenntnisse darüber, wie der Assembler unsere Programme in Maschinencode umsetzt.

Soll Ihr Programm die größtmögliche Effizienz aufweisen, dann sollten Sie sich einmal den Maschinencode in Abbildung 4.2 ansehen. Die Anzahl der Bytes eines Befehls steht nämlich in direktem Zusammenhang mit der Ausführungszeit. So benötigt z.B. die MOV-Anweisung, die einen direkten Wert im Speicher ablegt, 6 Bytes. Der 8088 verfügt aber über verschiedene Anweisungen, die speziell für den Zugriff auf das Akkumulator-Register, also AX oder AL, optimiert sind. Wenn Sie diese Anweisungen an geeigneter Stelle verwenden, können Sie in kritischen Programmen Zeit und Platz sparen.

Die letzten beiden Befehle in Abbildung 4.2 zeigen uns, wie ein direkter Wert im Segmentregister abgelegt wird. Zu diesem Zweck kann jedes beliebige Register, in unserem Fall das AX-Register, den Wert vorübergehend zwischenspeichern, bevor ihn unser Programm dann im Segmentregister ablegt.

Es gibt noch weitere zusätzliche Anweisungen zum Datentransport. Ein Beispielprogramm in Abbildung 4.3 zeigt diese Befehle.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.3  Data Movement instructions

1                                          PAGE    ,132
2                                          TITLE   Figure 4.3  Data Movement instructions
3       0000                       CODE    SEGMENT
4                                          ASSUME  CS:CODE,DS:CODE
5       0000                       EXDWORD LABEL   DWORD
6       0000                       EXWORD  LABEL   WORD
7       0000                       EXBYTE  LABEL   BYTE
8
9       0000  87 D9                        XCHG    BX,CX           ; Register BX <--> Register CX
10      0002  87 1E 0000 R                 XCHG    BX,EXWORD       ; Register BX <--> Memory
11      0006  93                           XCHG    AX,BX           ; Register AX <--> Register BX
12
13      0007  E4 20                        IN      AL,020H         ; Port 20 --> AL
14      0009  EC                           IN      AL,DX           ; Port (DX) --> AL
15      000A  E6 21                        OUT     021H,AL         ; AL --> Port 021H
16      000C  EE                           OUT     DX,AL           ; AL --> Port (DX)
17
18      000D  8D 36 0000 R                 LEA     SI,EXWORD       ; Address(EXWORD) --> SI
19      0011  C5 36 0000 R                 LDS     SI,EXDWORD      ; M(EXDWORD  --> SI
20                                                                 ; M(EXDWORD+2) --> DS
21      0015  C4 3E 0000 R                 LES     DI,EXDWORD      ; M(EXDWORD) --> DI
22                                                                 ; M(EXDWORD+2) --> ES
23
24      0019  9F                           LAHF                    ; Flags --> AH
25      001A  9E                           SAHF                    ; AH --> Flags
26
27      001B  D7                           XLAT    EXBYTE          ; M(BX+AL) --> AL
28
29      001C                       CODE    ENDS
30                                         END
```

Abbildung 4.3 Datentransportbefehle

Austausch

Der Befehl XCHG bewirkt ein einfaches Vertauschen des Inhalts von zwei Speicherstellen. Mit diesem Befehl können die Inhalte von zwei Registern oder von einem Register und einer Speicherstelle vertauscht werden. Ein Segmentregister kann allerdings nicht als Operand verwendet werden.

Der Tauschbefehl ersetzt dabei drei MOV-Befehle und benötigt auch keinen Zwischenspeicherplatz. Gäbe es diesen Befehl nicht, so würden wir in einem Programm, um die Register AX und BX zu vertauschen, drei MOV-Befehle benötigen. Zuerst müßten wir den Inhalt des Registers AX an eine Zwischenspeicherstelle transportieren, dann das Register BX nach AX transportieren und schließlich den Inhalt der Zwischenspeicherstelle wieder nach BX zurückspeichern. Die Anweisung XCHG bewältigt dies in einem einzigen Befehl.

Ein- und Ausgabe

Der 8088 hat für Ein- und Ausgabe jeweils eigene Befehle, nämlich IN für die Eingabe und OUT für die Ausgabe. Jedes Ein- und Ausgabegerät des IBM PC verfügt dabei über ein oder mehrere Register, die durch diese Ein-/Ausgabeanweisungen manipuliert werden können. Jedes Register in den verschiedenen Ein-/Ausgabegeräten ist einzeln adressierbar. Die Adressen unterscheiden sich aber von den normalen Speicheradressen des Systems. Der Adressbereich für die Ein-/Ausgabegeräte ist nämlich vom Adressbereich für den Arbeitsspeicher getrennt. Der 8088 verfügt über insgesamt 2^{16} oder 65.536 Ein-/Ausgabeadressen. Der IBM PC verwen-

det 512 dieser Adressen für den System-Ein-/Ausgabekanal bzw. für die verschiedenen Ein-/Ausgabeadapter. Weitere 256 Adressen werden auf der Systemplatine verwendet, um die dort befindlichen Ein-/Ausgabegeräte zu steuern.

Der IN-Befehl transportiert die Daten von einem Gerät in das AL-Register. Dabei kann die Adresse des jeweiligen Geräts auf zwei verschiedene Arten angegeben werden. Liegt die gewünschte Adresse im Bereich zwischen 0 und 255, so kann sie als Direktwert im Befehl angegeben werden. Ist die Adresse größer als 255, dann wird das gewünschte Gerät indirekt adressiert. In diesem Falle enthält das DX-Register die Ein-/Ausgabeadresse für den indirekten IN-Befehl. Mit dem DX-Register können übrigens alle Ein-/Ausgabegeräte indirekt adressiert werden, auch die mit einer Adresse unter 256.

Der OUT-Befehl arbeitet in ganz ähnlicher Weise, nur daß er den Inhalt des AL-Registers an das Ein-/Ausgabegerät abliefert. Auch die Adressen für den OUT-Befehl werden ebenso wie für den IN-Befehl spezifiziert.

Mit den Befehlen IN und OUT kann anstelle eines Bytes auch ein ganzes Wort von und nach den Ein-/Ausgabegeräten transportiert werden. Für einen Wortbefehl ist dann das AX-Register Quelle oder Ziel. Da aber der 8088 nur einen 1-Byte breiten externen Bus hat, akzeptieren auch die Ein-/Ausgabegeräte des IBM PC jeweils nur ein Byte für die Ein- oder Ausgabe. Dies bedeutet, daß Ein-/Ausgabeoperationen in Wortlänge auf dem IBM PC nicht ausgeführt werden können. Trotzdem könnten, da der 8086 über einen identischen Befehlssatz verfügt, Wortoperationen auf einem 8086-System von Wert sein.

Laden Effektive Adresse

Der Befehl „Laden Effektive Adresse" - LEA ist dem MOV-Befehl sehr ähnlich. Doch anstelle Daten von einer Speicherstelle in ein Register zu laden, lädt der Befehl LEA die Adresse dieser Daten in das Register. Da in den Befehlen des 8088 nur jeweils eine einzige Speicheradresse verwendet werden kann, muß als Ziel beim Befehl LEA immer ein Register angegeben werden. Dabei kann der Quelloperand in allen Adressierungsarten angegeben werden, die mittels des Mod-R/M-Bytes verfügbar sind.

In vielen Fällen entspricht der Befehl LEA einem MOV-Befehl mit Direktoperanden. Die Befehle

```
MOV     BX,OFFSET EXWORD
LEA     BX,EXWORD
```

führen völlig identische Operationen aus. Der erste Befehl ist ein MOV-Befehl, bei dem als Direktwert der Offset der Variablen EXWORD Verwendung findet. Die Angabe OFFSET teilt dem Assembler dabei mit, daß er das BX-Register mit dem Adressoffset der Variablen EXWORD laden soll (alle Adressen bestehen aus der Segmentadresse und dem Adressoffset). Der Befehl LEA dagegen berechnet die effektive Adresse der Variablen EXWORD und speichert sie im Register BX. In diesem Falle werden also von beiden Befehlen identische Operationen ausgeführt.

Würde unser Programm jedoch in das BX-Register die Adresse des zehnten Bytes eines Arrays laden, das über das DI-Register adressiert wird, würde der Befehl LEA wie folgt aussehen:

```
LEA    BX,10[DI]
```

Der 8088 würde dann die Adresse über die Mod-R/M-Information berechnen, wie er es auch für einen MOV-Befehl tun würde. Dann speichert er allerdings den berechneten Adressoffset in das BX-Register, aber nicht die Daten, die sich an dieser Stelle befinden. In diesem Fall gibt es keine MOV-Anweisung mit direkten Werten, die eine gleiche Funktion ausführen könnte. Da die Adresse nämlich zur Assemblierungszeit nicht bekannt ist, gibt es für den Assembler keine Möglichkeit, diese Adresse als Direktwert zu bestimmen.

Laden Pointer

Da der Adressmechanismus des 8088 sowohl Segmentadresse als auch Adressoffset für jede Variable zur Verfügung stellt, ist es wünschenswert, die Gesamtheit dieser Information mit einem einzigen Befehl zu laden. Die Anweisungen LDS und LES ermöglichen dies. Dabei lädt die Anweisung

```
LDS    SI,EXDWORD
```

das Registerpaar DS:SI mit Segmentadresse und Adressoffset, die in der Variablen EXDWORD enthalten sind. Der Befehl LDS lädt dabei das SI-Register mit dem Offsetwert, der sich an der Adresse EXDWORD befindet, und das DS-Register mit dem Segmentwert, der sich an der Adresse EXDWORD+2 befindet. Der Befehl LDS lädt also je zwei 16-Bit-Register mit einem Zeigerwert auf eine beliebige Speicherstelle. Da der Befehl sowohl Segment- als auch Offset-Register lädt, kann der solchermaßen adressierte Punkt vom Programm sofort angesprochen werden. Wir können den Segmentwert und den Offsetzeiger zur Assemblierungszeit mit der Angabe DD festlegen. Die Angabe DD erzeugt nämlich ein 32 Bit langes Datenfeld. Ist der Operand einer DD-Anweisung nun ein Adressausdruck, so enthalten die beiden Wortfelder jeweils Segment und Adressoffset im gleichen Format, wie es die Befehle LDS und LES benötigen.

Der Befehl LES ist identisch zum Befehl LDS, mit der Ausnahme, daß er anstelle des DS-Registers das ES-Register lädt. Es gibt dagegen keine Befehle, die in der Lage sind, Segment- und Offsetwerte in einem einzigen Zug zu speichern. Ein Programm muß also Zeigerwerte in zwei getrennten Wortoperationen speichern. Wir können dies hinnehmen, da in einem Programm Zeigerwerte wesentlich öfter gelesen als gespeichert werden. Normalerweise setzt ein Programm nämlich Adresszeigerwerte einmal zu Beginn der Abarbeitung, und setzt sie vielleicht irgendwann zurück, wenn sich die Verarbeitungsart verändert. In der Zwischenzeit wird dieser Zeigerwert vermutlich recht häufig verwendet, aber eben nur gelesen. In späteren Kapiteln werden wir einige Beispiele zeigen, in denen Adresszeigerwerte sowohl gespeichert als auch gelesen werden.

Flagtransport

Die Anweisungen LAHF und SAHF sind im Befehlssatz des 8088 hauptsächlich deswegen enthalten, um Kompatibilität zum Befehlssatz des 8080 herzustellen. Der Befehl LAHF nimmt dabei die niederwertigen acht Bit des Flagregisters — die Flags, die identisch mit den Flags des 8080 sind — und speichert sie in das AH-Register. Der Befehl SAHF arbeitet in entgegengesetzter Richtung. Der im AH-Register enthaltene Wert wird also an die Stelle der niederwertigen acht Bits des Flagregisters gespeichert.

Wollen wir ein Programm vom 8080 auf den 8088 übernehmen, so benötigen wir diese beiden Befehle. Wir brauchen sie nämlich, um die Akkumulator-Stack-Operationen des 8080 auf die Stackoperationen des 8088 abzubilden. Wir werden zusätzlich den Befehl SAHF auch noch im Kapitel 7 verwenden, um für bestimmte bedingte Sprungbefehle Werte in die Flags zu laden.

Datenumsetzung

Der Umsetzungsbefehl XLAT setzt Daten von einer Darstellungsart in eine andere um. Der Befehl ersetzt dabei den Wert im AL-Register durch einen Wert, der sich in einer Tabelle befindet, die durch das BX-Register adressiert wird. Abbildung 4.4 zeigt schematisch den Ablauf dieses Befehls. Das BX-Register, zusammen mit einem ausgewählten Segmentregister, beinhaltet den Startpunkt der entsprechenden Umsetzungstabelle im Speicher. Der Befehl addiert nun den Inhalt des AL-Registers, einen Wert zwischen 0 und 255, auf die Anfangsadresse dieser Tabelle. Der Inhalt der solchermaßen adressierten Speicherstelle wird dann wieder in das AL-Register übertragen. Der Befehl XLAT ersetzt also Daten über eine sogenannte Umsetzungstabelle.

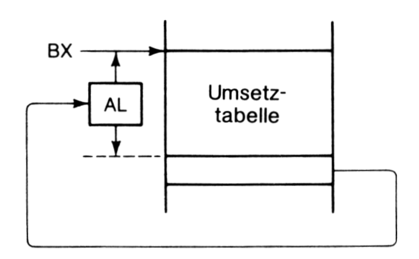


Abbildung 4.4 XLAT-Befehl

Ein sinnvoller Anwendungsbereich für diesen Befehl ergibt sich im Kodieren und Dekodieren von Textdaten. Mittels dieses Befehles können wir in einem Programm nämlich einen einfachen Ersetzungsmechanismus aufbauen. In einem Beispielprogramm verschlüsseln wir dazu die zehn ASCII-Zeichen 0 bis 9 für Übertragungszwecke. Diese Technik könnte z.B. verwendet werden, um empfindliche Daten vor der Übertragung von einer Maschine auf eine andere zu verschlüsseln. Auf der

Empfängerseite werden dann die so verschlüsselten Daten durch ein entsprechendes Programm wieder entschlüsselt. Abbildung 4.5 zeigt eine solche Ersetzungstabelle.

Original	Gesendet	(b) Empfangen	Korrekt
0	5	0	7
1	7	1	3
2	9	2	8
3	1	3	4
4	3	4	9
5	6	5	0
6	8	6	5
7	0	7	1
8	2	8	6
9	4	9	2
(a)		(b)	

Abbildung 4.5 Umsetztabelle für ASCII-Ziffern

In Abbildung 4.5 sehen wir zwei Übersetzungstabellen, eine für den Sender und eine für den Empfänger. Um den Wert 5 zu übertragen sucht unser Programm diesen Wert in der Sendetabelle (a). Es findet dort den Wert 6 und sendet ihn. Wird dieser Wert empfangen, sieht unser Entschlüsselungsprogramm an der Stelle 6 in der Empfangstabelle (b) nach und entdeckt wiederum den richtigen Wert, nämlich 5.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 4.6 Translate Example

 1                                              PAGE    ,132
 2                                              TITLE   Figure 4.6 Translate Example
 3      0000                            CODE    SEGMENT
 4                                              ASSUME  CS:CODE,DS:CODE
 5
 6                                      ;     This program reads the value from input port 040H,
 7                                      ;     and converts it using the encryption table.
 8                                      ;     Since the same routine does both encryption and decryption,
 9                                      ;     it uses the double word pointer TABLE_POINTER to point
10                                      ;     to the correct translation table.
11                                      ;     The using program must set this pointer to the correct
12                                      ;     table prior to calling the subroutine.
13
14      0000                            TRANSLATE       PROC    NEAR            ; Subroutine called TRANSLATE
15
16      0000  E4 40                             IN      AL,040H                 ; Read in the code value
17      0002  2C 30                             SUB     AL,"0"                  ; Convert to zero origin
18      0004  C5 1E 000A R                      LDS     BX,TABLE_POINTER        ; (DS,BX) now points at table
19      0008  D7                                XLAT    XMIT_TABLE              ; Convert number
20      0009  C3                                RET
21
22      000A  000E ---- R               TABLE_POINTER   DD      XMIT_TABLE
23      000E  35 37 39 31 33 36         XMIT_TABLE      DB      '5791368024'
24            38 30 32 34
25      0018  37 33 38 34 39 30         RECV_TABLE      DB      '7384905162'
26            35 31 36 32
27
28      0022                            TRANSLATE       ENDP
29      0022                            CODE    ENDS
30                                              END
```

Abbildung 4.6 Umsetzbeispiel

Abbildung 4.6 zeigt ein Unterprogramm zur Datenverschlüsselung. Die Umsetzungsroutine liest dabei den Orginalwert von einem Ein-/Ausgabeport und gibt den kodierten bzw. dekodierten Wert an das aufrufende Programm im AL-Register zurück. Das gleiche Unterprogramm ist also in der Lage, sowohl das Kodieren als auch das Dekodieren zu übernehmen, indem ganz einfach die Umschlüsselungstabelle ausgetauscht wird.

Unser Unterprogramm liest dazu als erstes die Daten vom Eingabeport 40H in das AL-Register. Sodann wird der Wert des ASCII-Zeichens „0" von den erhaltenen Daten subtrahiert, um die Zahlenreihe bei 0 beginnen zu lassen. Das bedeutet, daß das Zeichen „0" im AL-Register nun auch den Wert 0 erhält, das Zeichen „1" wird eine 1 sein, und so weiter. Der Befehl LDS lädt sodann den Zeiger auf die gewünschte Umschlüsselungstabelle in das Registerpaar DS:BX. Laden wir diesen Pointer von einer Speicherstelle — TABLE_POINTER in unserem Beispiel — so können wir jede beliebige Umschlüsselungstabelle verwenden. In unserem Programm gibt es allerdings nur zwei Tabellen. Eine, bezeichnet als XMIT_TABLE, ist für die Sendung gedacht und entspricht Abbildung 4.5 (a). Die Empfangstabelle, bezeichnet als RECV_TABLE, entspricht Abbildung 4.5 (b). Bevor wir nun unser Unterprogramm aufrufen, muß das Hauptprogramm die Adresse der gewünschten Tabelle in die Variable TABLE_POINTER gespeichert haben. Würde unser Hauptprogramm z.B. Daten erhalten, so müßten wir die Adresse von RECV_TABLE in die Variable TABLE_POINTER speichern. Halten wir fest, daß mit unserem Unterprogramm praktisch jede Umsetzung möglich ist, da das jeweils aufrufende Programm die Umsetzungstabelle erst festlegt.

Der Befehl XLAT führt nun die Umsetzung aus, wobei das Registerpaar DS:BX auf die Umschlüsselungstabelle zeigt. Das AL-Register enthält einen Wert zwischen 0 und 9. XLAT addiert nun diesen Wert auf den Zeiger und lädt den umgesetzten Wert in das Register AL. Der Befehl RET gibt dann die Steuerung wieder zurück an das aufrufende Programm.

Eine weitere allgemein übliche Anwendung des Befehls XLAT ist die Umsetzung von Zeichen aus einer Maschinendarstellung in eine andere. So arbeitet der IBM PC beispielsweise in ASCII, doch die meisten IBM-Großrechner verwenden EBCDIC (Extended Binary-Coded Decimal Interchange Code). Um mit solchen Maschinen in Verbindung treten zu können, muß ein Programm die Werte der einzelnen Zeichen umsetzen. Und der Befehl XLAT ist dafür hervorragend geeignet.

Zusammenfassend können wir sagen, daß der Befehl XLAT ein sehr mächtiges Werkzeug ist, wenn wir Zeichen oder Bytes umsetzen wollen. Trotzdem ist er eine selten verwendete Anweisung, da es nur wenig Gelegenheit gibt, wo man wirklich Nutzen aus ihm ziehen kann. Doch behalten wir ihn in Erinnerung für den Fall, daß wir ihn wirklich einmal benötigen.

Stackbefehle

In Kapitel 3 besprachen wir bereits, wie auf dem 8088 der Stack implementiert ist. Dabei wird der Stack über das Registerpaar SS:SP adressiert. Speichern wir Daten in den Stack, führt dies dazu, daß der Stack in Richtung auf die niedrigeren Speicheradressen wächst. Der Stack dient außerdem als Speicherstelle für die Rückkehradressen von Unterprogrammen. Im weiteren werden wir nun einige Anweisungen besprechen, die sich direkt auf den Stack beziehen.

Abbildung 4.7 zeigt ein Programm mit den möglichen Stackanweisungen. Die Befehlsbezeichnungen sind einfach, PUSH und POP, gefolgt von dem als Operand

gewünschten Register. Die einzige Ausnahme ist PUSH oder POP mit dem Flagregister, wo wir dann PUSHF und POPF schreiben müssen. Der Inhalt jeder Speicherstelle, die in einem normalen Programm adressierbar ist, kann ebenso in den Stack geschrieben oder auch wieder herausgelesen werden.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.7 Stack operations

1                                       PAGE    ,132
2                                       TITLE   Figure 4.7 Stack operations
3       0000                    CODE    SEGMENT
4                                       ASSUME  CS:CODE,DS:CODE
5       0000                    EXWORD  LABEL   WORD
6
7       0000  50                        PUSH    AX              ; Push a register
8       0001  56                        PUSH    SI
9       0002  0E                        PUSH    CS              ; Can push segment registers
10      0003  FF 36 0000 R              PUSH    EXWORD          ; Can push memory locations
11
12      0007  8F 06 0000 R              POP     EXWORD          ; Can pop anything that pushes
13      000B  07                        POP     ES              ; Pop need not match push
14      000C  5F                        POP     DI
15      000D  5B                        POP     BX
16
17      000E  9C                        PUSHF                   ; Different mnemonic for flags
18      000F  9D                        POPF
19
20                              ;----- Example showing paramter passing
21
22      0010  50                        PUSH    AX              ; Save 4 parameters on stack
23      0011  53                        PUSH    BX
24      0012  51                        PUSH    CX
25      0013  52                        PUSH    DX
26      0014  E8 0017 R                 CALL    SUBROUTINE      ; Transfer control
27                              ;       ...                     ; Continue processing
28
29      0017                    SUBROUTINE      PROC    NEAR
30
31      0017  8B EC                     MOV     BP,SP           ; Establish stack address
32      0019  8B 46 02                  MOV     AX,[BP+2]       ; Get last parm (DX)
33      001C  8B 5E 04                  MOV     BX,[BP+4]       ; Get 3rd parm (CX)
34      001F  8B 4E 06                  MOV     CX,[BP+6]       ; Get 2nd parm (BX)
35      0022  8B 56 08                  MOV     DX,[BP+8]       ; Get first parm (AX)
36                              ;       ...
37      0025  C2 0008                   RET     8               ; Return, and discard parms
38      0028                    SUBROUTINE      ENDP
39      0028                    CODE    ENDS
40                                      END
```

Abbildung 4.7 Stackbefehle

Alle Stackbefehle des 8088 arbeiten in Wortbreite. Alle Daten, die in den Stack gespeichert oder die aus dem Stack gelesen werden, müssen ein oder mehrere Wörter lang sein. Byte-Operationen können also mit PUSH oder POP nicht durchgeführt werden. Muß ein Programm beispielsweise den Inhalt des AL-Registers in den Stack sichern, so muß es das gesamte AX-Register in den Stack speichern, da es keine Möglichkeit gibt, das AL-Register einzeln zu sichern.

Der Hauptzweck des Stacks ist, Informationen zeitweilig zu speichern. Wir haben bereits gesehen, daß der Stack dazu benutzt wird, die Rückkehradressen von Unterprogrammen zu speichern. Wir können aber durchaus auch Daten im Stack speichern. Benötigen wir in einem Programm beispielsweise ein Register, wollen aber die Daten in dem Register nicht zerstören, so können wir dieses Register zeitweilig in den Stack speichern. Die Daten sind also im Stack sicher aufgehoben, und später können wir sie bei Bedarf wieder auslesen. Wir können in einem Programm beispielsweise Daten von der Ein-/Ausgabeadresse 3DAH lesen wollen, das DX-Register würde aber an dieser Stelle einen wichtigen Wert enthalten.

```
PUSH    DX
MOV     DX,3DAH
IN      AL,DX
POP     DX
```

Mit der gezeigten Befehlsfolge sichern wir den Inhalt des DX-Registers in den Stack, solange wir das DX-Register für den IN-Befehl benötigen.

Im allgemeinen verwenden wir den Stack auch zu Beginn eines Unterprogramms. In einem Unterprogramm sollten wir es nämlich vermeiden, die Inhalte von Registern zu verändern. So speichern wir die Inhalte all der Register, die wir für temporäre Berechnungen und Adresswerte benötigen, in den Stack, bevor wir das Unterprogramm ausführen. Dann, nach Ausführung des Unterprogramms, stellen wir den Inhalt der Register durch den POP-Befehl wieder her.

Erinnern wir uns, daß der Stack eine LIFO-Datenstruktur ist. Das letzte in den Stack gespeicherte Wort wird als erstes wieder ausgelesen. Weist unser Programm die Befehlsfolge

```
PUSH    BX
PUSH    CX
...
POP     BX
POP     CX
```

auf, so ist der Erfolg, daß die Inhalte der Register BX und CX vertauscht werden. Benützen wir nämlich in einem PUSH-Befehl das BX-Register, so bedeutet dies noch lange nicht, daß ein POP-Befehl, der ebenfalls wieder das BX-Register anspricht, auch den gleichen Wert wieder zurückerhält. Weiterhin ist wichtig, daß PUSH und POP sich immer die Waage halten müssen – d.h., für jeden PUSH-Befehl muß es auch einen passenden POP-Befehl geben. Genau wie bei Rechenoperationen, bei denen die Zahl der öffnenden Klammern nicht mit der Zahl der schließenden übereinstimmt, werden wir auch bei PUSH- und POP-Befehlen, die sich nicht die Waage halten, falsche Resultate erhalten. Schlimmer noch, da der 8088 auch die Rückkehradressen für Unterprogramme in den Stack speichert, wird ein Ungleichgewicht bei PUSH- und POP-Befehlen häufig dazu führen, daß ein Unterprogramm auf einen Datenwert zurückkehrt und nicht an die Stelle, wo es eigentlich weitermachen sollte. Das führt dann im allgemeinen dazu, daß der Prozessor ein Programm ausführt, das der Programmierer so nie wollte. Es ist deshalb beinahe zwingend, die Stackoperationen so anzulegen, daß sie sich die Waage halten. Seien Sie besonders sorgfältig, wenn ein Programm bedingte Sprünge zusammen mit Stackoperationen enthält. Man kann dabei nämlich leicht eine Ausführungsmöglichkeit übersehen, so daß die Stackbefehle unausgeglichen sein könnten.

Neben der Möglichkeit, Daten zu sichern, können wir in einem Programm den Stack auch als Zwischenstation für Datentransporte verwenden. So gibt es z.B. keinen direkten Befehl, der in der Lage ist, Daten von einem Segmentregister in ein anderes zu transportieren. Es ist deshalb nötig, beim Transport der Daten diese zuerst in einem Zwischenregister unterzubringen. Dies wird dann eine Folge von zwei Befehlen wie:

```
MOV   AX,CS   ; Bringe den CS-Wert ins AX-Register
MOV   DS,AX   ; Speichere den Wert im DS-Register
```

Jeder dieser beiden Befehle ist mehrere Bytes lang, und außerdem zerstören wir den Inhalt des AX-Registers. Ein anderer Weg könnte sein:

```
PUSH  CS   ; CS-Register in den Stack
POP   DS   ; Speichere den Wert im DS-Register
```

Das Ergebnis der Befehlsfolgen ist gleich. In beiden Fällen wird das DS-Register mit dem im CS-Register enthaltenen Wert geladen. Der Befehlscode im zweiten Beispiel ist allerdings nur zwei Bytes lang und benötigt keinen Zwischenspeicher. Allerdings benötigen diese beiden Befehle eine längere Ausführungszeit, da zum Lesen und Schreiben der Daten in den Stack jeweils zusätzliche Speicherzyklen notwendig sind. Dies ist also auch ein Beispiel dafür, wie man die Programmgröße auf Kosten der Ausführungsgeschwindigkeit verringern kann.

Übergabe von Parametern

Der Stack kann auch als geeignetes Mittel dienen, um Informationen von und nach Unterprogrammen zu übermitteln. Normalerweise übergeben wir Parameter an ein Unterprogramm, indem wir sie in einem Register hinterlegen. Es kann jedoch auch Fälle geben, in denen die Anzahl der Parameter die möglichen Register übersteigt. In diesen Fällen könnte unser Programm die Parameter in den Stack schreiben, bevor es den CALL-Befehl ausführt. Wie wir später in Kapitel 10 sehen werden, ist der Stack außerdem die einzige mögliche Methode, Parameter an Assemblerroutinen zu übergeben, wenn sie von höheren Sprachen wie BASIC oder FORTRAN aufgerufen werden.

Unser Unterprogramm nützt nun diese Parameter im Stack auf eine sehr wirksame Weise. Normalerweise kann ein Programm auf Daten im Stack nur dann zugreifen, wenn es diese Daten mittels POP aus dem Stack liest. Anstelle dessen können wir im Unterprogramm das BP-Register als Zeiger auf den Stackbereich verwenden. Übergibt ein Programm nun Parameter im Stack, so ist eine der ersten Anweisung im Unterprogramm

```
MOV   BP,SP
```

wodurch das BP-Register auf den aktuellen Wert des Stackpointers gesetzt wird. Da das BP-Register ein Adressregister ist, kann es vom Unterprogramm bei der Berechnung von Adressen verwendet werden. Das bedeutet, daß alle Parameter als Displacement über das BP-Register angesprochen werden können.

Den Entwicklern des 8088 schwebte diese Methode der Parameterübergabe sicher auch vor, denn das BP-Register benützt das Stack-Segmentregister (SS) als Standardsegmentregister, wenn auf Daten zugegriffen wird. Für alle anderen Datenreferenzen benützt der Prozessor normalerweise das DS-Register. Da sich der Stack im Stacksegment befindet, zeigt das Registerpaar SS:BP natürlicherweise immer auf die gerade aktuelle Adresse im Stack. Abbildung 4.7 zeigt ein Unterprogramm,

das das BP-Register zur Adressierung der im Stack übergebenen Parameter verwendet. Im Beispiel schreibt das Hauptprogramm vier Wortparameter in den Stack, bevor es das Unterprogramm aufruft. Das Unterprogramm setzt nun das BP-Register so, daß es auf die Daten im Stack zeigt. Halten wir fest, daß wir beim Displacement zum Zugriff auf die im Stack gespeicherten Daten mit berücksichtigen müssen, daß auch die Rückkehradresse in das aufrufende Hauptprogramm durch den CALL-Befehl im Stack gespeichert ist.

In unserem Beispielsunterprogramm ist der Wert am Anfang des Stacks die Rückkehradresse in das aufrufende Hauptprogramm. Das BP-Register enthält den Adressoffset auf diese Speicherstelle. Zwei Bytes davon entfernt befindet sich der letzte Parameter, der in den Stack gespeichert wurde, das DX-Register. In 2-Byte-Abständen folgen darauf die Register CX, BX und AX. So ist die korrekte Adresse, um den Parameter zu erhalten, der sich im DX-Register befand, [BP+2], und die anderen Werte folgen in 2-Byte-Abständen. Außerdem wandert der ursprünglich im DX-Register enthaltene Wert jetzt in das AX-Register, CX nach BX, usw.

Die Parameterübergabe ist nicht der einzige Punkt, an dem ein Unterprogramm das BP-Register zur Stackadressierung verwenden kann. Ein Unterprogramm könnte so lange und verwickelt sein, daß es schwierig wäre, alle notwendigen Parameter während des Ablaufs in Registern zu halten. Das Speichern all dieser Werte in den Stack und das Setzen des BP-Registers als Zeiger auf den Stackbereich löst dieses Problem.

Viele Unterprogramme benötigen während ihres Ablaufs auch einigen lokalen Speicher. Das Unterprogramm könnte diesen Speicher dynamisch aus dem Stack beziehen. Dazu ist es nötig, nach jedem Unterprogrammaufruf die Größe des benötigten Speicherbereichs vom Stackpointer zu subtrahieren. Da der Stack in Richtung auf die niedrigeren Speicheradressen wächst, bedeutet das Subtrahieren eines Werte vom SP-Register, daß der Stack um genau diese Anzahl von Bytes wächst — abgesehen davon, daß der so gewonnene Datenbereich undefinierte Werte enthält. Dann können wir das BP-Register benützen, um den neuen Speicherplatz zu adressieren. Wenn es an der Zeit ist, das Unterprogramm wieder zu verlassen, addieren wir den entsprechenden Wert wieder auf den Stackpointer. Auf diese Weise erhält der Stackpointer wieder seinen ursprünglichen Wert. Dynamische Speicherzuteilung bedeutet hier, daß unser Programm den Speicher nur dann verwendet, wenn er zum Ablauf benötigt wird, der Speicher also nicht grundsätzlich für diese Zwecke reserviert ist. Damit ist es möglich, Programme laufen zu lassen, die sonst wegen des beschränkten Speicherplatzes nicht auf unserem Rechner hätten laufen können. Das schönste daran ist, daß wir dazu nicht einmal eine komplizierte Speicherverwaltung benötigen. Die Struktur des Stacks hält dazu alles unter Kontrolle.

Die Rückkehr aus dem Unterprogramm in Abbildung 4.7 zeigt eine weitere Möglichkeit des Befehlssatzes des 8088. Die Rückkehranweisung aus einem Unterprogramm (RET) kann einen Operanden haben. Dieser Operand ist ein Wert, den der Prozessor auf den Stackpointer addiert, nachdem er die Rückkehradresse aus dem Stack gelesen hat. Unser Beispiel verwendet dabei den Wert 8. Das bedeutet, daß acht Bytes oder vier Worte nach der Rückkehr aus dem Unterprogramm aus dem

Stack entfernt werden. Diese Werte verschwinden und tauchen nie wieder auf. Der Effekt ist derselbe, wie wenn wir nach der Rückkehr aus dem Unterprogramm viermal irgendwelche Daten aus dem Stack lesen würden. Das aufrufende Programm muß also die übergebenen Parameter nicht nach Rückkehr aus dem Unterprogramm wieder einzeln aus dem Stack lesen und sie dann vernichten. Die Rückkehranweisung hat dies bereits automatisch für uns erledigt.

Diese Methode des Datenlöschens aus dem Stack funktioniert natürlich nur für Parameter, die das aufrufende Programm selbst in den Stack gestellt hat. Das Unterprogramm seinerseits muß jeden dynamisch zugeordneten Speicherplatz wieder aus dem Stack löschen, bevor es in das Hauptprogramm zurückkehrt. Im Unterprogramm darf dies nicht durch die RET-Anweisung geschehen, sondern muß explizit ausgeführt werden, da der Datenbereich zwischen der gerade aktuellen Spitze des Stacks und der Rückkehradresse liegt.

Ein Unterprogramm kann auch Informationen an das aufrufende Programm im Stack zurückgeben. Übergibt das aufrufende Programm nämlich Werte im Stack, so kann das Unterprogramm diese Werte modifizieren und sie so wieder zurückgeben. Das aufrufende Programm kann sie nach der Rückkehr aus dem Unterprogramm dann aus dem Stack lesen. Gibt unser Unterprogramm nur einen Parameter zurück, wurde aber mit drei Parametern aufgerufen, so ist die Anweisung RET 4 das geeignete Mittel hierfür. Dadurch werden die letzten beiden Parameter gestrichen und der einzige gewünschte Rückkehrparameter bleibt im Stack stehen. Wenn wir später in Kapitel 10 Assemblerprogramme mit Programmen in höheren Programmiersprachen kombinieren, übergibt das Hauptprogramm die Parameter in den Stack. Doch diese Parameter werden ihrerseits nur die Adressen der Daten sein und nicht die Werte selbst. Das bedeutet, daß unser Assembler-Unterprogramm keine Werte im Stack zurückgeben muß und wir deshalb bei der Rückkehr aus dem Unterprogramm alle Parameter aus dem Stack löschen sollten.

Arithmetische Befehle

Die arithmetischen Befehle eines jeden Prozessors sind diejenigen, die die meiste Aufmerksamkeit auf sich ziehen. Jeder möchte nämlich rechnen, und diese Anweisungen sind dazu gedacht. Obwohl sie nur sehr wenige sind, übernehmen sie doch den Hauptteil der Datenbehandlung im Prozessor. Bei der Programmausführung sind allerdings die arithmetischen Befehle nur ein ganz geringer Teil der insgesamt ausgeführten Anweisungen.

Die bisher besprochenen MOV-Befehle behandelten bereits viele der Eigenschaften des Befehlssatzes des 8088. Doch es gibt noch einige zusätzliche Feinheiten, die wir besser bei den arithmetischen Befehlen erläutern.

Addition

Der ADD-Befehl führt die Addition des Zweierkomplements seiner Operanden aus. Nach der Addition der beiden Operanden legt der Prozessor das Ergebnis im ersten

Operanden ab. Der zweite Operand bleibt unverändert. Außerdem verändert der Befehl das Flagregister in Abhängigkeit vom Ergebnis der Addition. Der Befehl

ADD AX,BX

addiert den Inhalt des Registers BX auf den Inhalt des Registers AX und hinterläßt das Ergebnis im Register AX. Das Flagregister teilt uns dann mit, ob das Ergebnis 0 oder negativ war, ob es eine gerade Parität aufwies, und ob ein Überlauf oder Übertrag auftrat.

Abbildung 4.8 faßt die Wirkungen des ADD-Befehls zusammen. Dabei gibt es zwei Formen der Addition, 8-Bit und 16-Bit. Halten wir dabei fest, daß an den verschiedenen Arten der Addition jeweils verschiedene Register beteiligt sind. Der Assembler sorgt dafür, daß die Operanden zueinander passen. So darf ein Byteregister (bei-

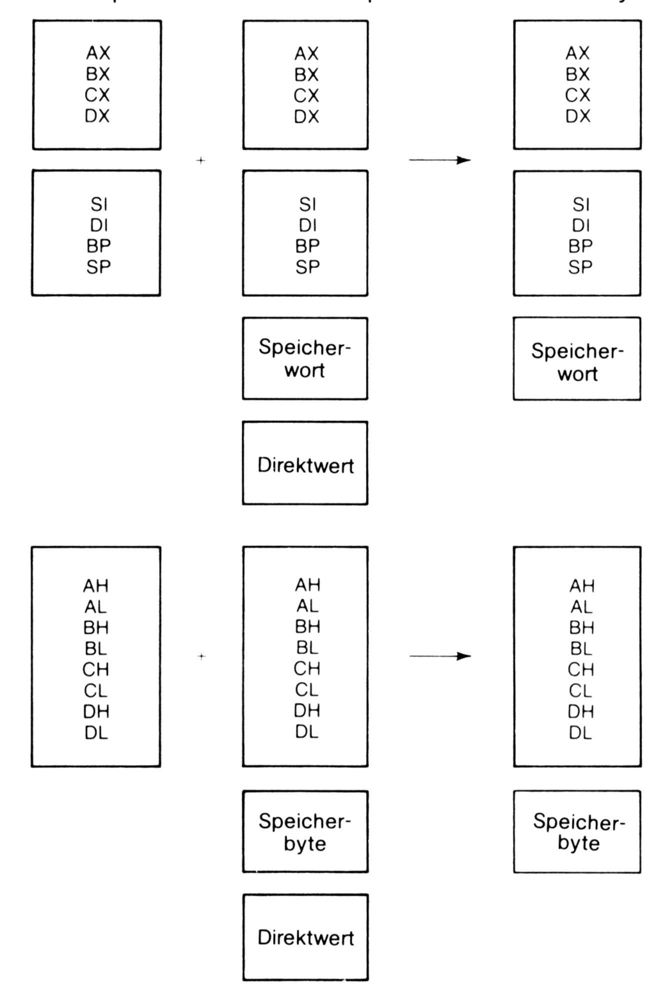


Abbildung 4.8 Additionsbefehle

spielsweise das Register CH), nicht auf den Inhalt einer Speicherstelle addiert werden, die nicht als Byte definiert ist. Wenn einer der Operanden eine Speicherstelle ist, so kann er entweder der Ziel- oder der Quelloperand sein. Dies erlaubt es auch, ein Register auf eine Speicherstelle zu addieren und das Ergebnis dann in der Speicherstelle zu halten. Auch ein Direktwert kann einer der Operanden sein. In der Assemblerliste in Abbildung 4.9 sehen wir einige der arithmetischen Anweisungen.

Der Befehl ADC (Add with Carry) entspricht dem Befehl ADD mit der Ausnahme, daß das Carryflag in die Addition miteinbezogen wird. Für jede Form der ADD-Anweisung gibt es auch eine vergleichbare ADC-Anweisung.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 4.9  Arithmetic Instructions

 1                                              PAGE    ,132
 2                                              TITLE Figure 4.9  Arithmetic Instructions
 3      0000                            CODE    SEGMENT
 4                                              ASSUME  CS:CODE,DS:CODE
 5      0000                            EXBYTE  LABEL   BYTE
 6      0000                            EXWORD  LABEL   WORD
 7
 8      0000  03 1E 0000 R                      ADD     BX,EXWORD        ; BX <- BX + [EXWORD]
 9      0004  29 0E 0000 R                      SUB     EXWORD,CX        ; [EXWORD] <- EXWORD - CX
10      0008  12 3E 0000 R                      ADC     BH,EXBYTE        ; BH <- BH + [EXBYTE] + Carry
11      000C  18 0E 0000 R                      SBB     EXBYTE,CL        ; [EXBYTE] <- [EXBYTE] - CL - Carry
12      0010  F7 1E 0000 R                      NEG     EXWORD           ; [EXWORD] <-  -[EXWORD]
13      0014  FE 06 0000 R                      INC     EXBYTE           ; [EXBYTE] <- [EXBYTE] + 1
14      0018  4E                                DEC     SI               ; SI <- SI - 1
15
16      0019  81 C7 00C8                        ADD     DI,200           ; DI <- DI + 200
17      001D  83 EC 64                          SUB     SP,100           ; SP <- SP - 100
18      0020  83 D1 0A                          ADC     CX,10            ; CX <- CX + 10 + Carry
19      0023  83 1E 0000 R 14                   SBB     EXWORD,20        ; [EXWORD] <- [EXWORD] - 20 - Carry
20
21      0028  3B C3                             CMP     AX,BX            ; Set flags for AX - BX
22      002A  81 FE 01F4                        CMP     SI,500           ; Set flags for SI - 500
23
24      002E  F6 26 0000 R                      MUL     EXBYTE           ; AX <- AL * [EXBYTE]
25      0032  F7 EB                             IMUL    BX               ; DX:AX <- AX * BX
26      0034  F7 36 0000 R                      DIV     EXWORD           ; AX <- DX:AX / [EXWORD]
27      0038  F6 FD                             IDIV    CH               ; AL <- AX / CH
28
29      003A  27                                DAA                      ; Decimal Adjust Addition
30      003B  2F                                DAS                      ; Decimal Adjust Subtraction
31      003C  37                                AAA                      ; ASCII Adjust Addition
32      003D  3F                                AAS                      ; ASCII Adjust Subtraction
33      003E  D4 0A                             AAM                      ; ASCII Adjust Multiplication
34      0040  D5 0A                             AAD                      ; ASCII Adjust Division
35      0042  98                                CBW                      ; AX <- Sign extend AL
36      0043  99                                CWD                      ; DX:AX <- Sign extend AX
37
38      0044                            CODE    ENDS
39                                              END
```

Abbildung 4.9 Arithmetische Befehle

Alle Additionsbefehle, sowohl ADD wie auch ADC, setzen das Carryflag, wenn es einen Übertrag über das höchstwertige Bit hinaus gibt. Eine normale ADD-Anweisung addiert dabei die beiden Operanden ohne Rücksicht auf das Carryflag. Der Befehl ADC dagegen berücksichtigt das Carryflag bei der Addition. Ist dabei der Inhalt des Carryflags 0, so entspricht das Ergebnis einer ADC-Operation dem eines normalen ADD-Befehls. Ist das Carrybit 1, so ist das Ergebnis um 1 höher als das einer normalen Addition. So kann ein Programm das Carryflag auch für Rechenoperationen von höherer Genauigkeit verwenden. Abbildung 4.10 zeigt beispielsweise die Addition von 32-Bit Zahlen. Dabei werden die beiden Zahlen VALUE1 und VALUE2 aufeinander addiert und das Ergebnis an die Stelle VALUE2 abgelegt. Halten wir dabei fest, daß vor der Operation einer der beiden Operanden in ein Register gestellt werden muß. In der ersten Addition verwenden wir den ADD-Befehl, da der

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.10 Multiple Precision Example

 1                                        PAGE    ,132
 2                                        TITLE   Figure 4.10 Multiple Precision Example
 3        0000                    CODE    SEGMENT
 4                                        ASSUME  CS:CODE,DS:CODE
 5
 6        0000  ????????          VALUE1  DD      ?       ; 32 bit data area
 7        0004  ????????          VALUE2  DD      ?
 8
 9                                ;----- This code fragment adds two 32 bit numbers
10
11        0008  A1 0000 R                 MOV     AX,WORD PTR VALUE1
12        000B  01 06 0004 R              ADD     WORD PTR VALUE2,AX      ; Perform low 16 bit addition
13        000F  A1 0002 R                 MOV     AX,WORD PTR VALUE1+2
14        0012  11 06 0006 R              ADC     WORD PTR VALUE2+2,AX    ; High 16 bit addition
15
16                                ;----- This code fragment subtracts two 32 bit numbers
17
18        0016  A1 0000 R                 MOV     AX, WORD PTR VALUE1
19        0019  29 06 0004 R              SUB     WORD PTR VALUE2,AX      ; Subtract low order portion
20        001D  A1 0002 R                 MOV     AX, WORD PTR VALUE1 + 2
21        0020  19 06 0006 R              SBB     WORD PTR VALUE2 + 2, AX ; Subtract high order portion
22
23        0024                    CODE    ENDS
24                                        END
```

Abbildung 4.10 Rechnen mit höherer Genauigkeit

aktuelle Wert des Carryflags zu diesem Zeitpunkt noch bedeutungslos ist. Nach dieser Operation führen wir die zweite Addition durch, und zwar unter Berücksichtigung des Carryflags aus der vorhergehenden Addition über die Anweisung ADC. Dies ist auch ein gutes Beispiel dafür, warum der MOV-Befehl keine Flags verändern darf. Würden nämlich durch den MOV-Befehl die Flags verändert, wäre die zweite Addition wesentlich schwieriger durchzuführen.

Subtraktion

Die Subtraktionsanweisungen SUB und SBB entsprechen vollkommen den Additionsanweisungen, außer daß sie an Stelle einer Addition eine Subtraktion durchführen. Sie können das Beispiel in Abbildung 4.8 auch für eine Subtraktion verwenden, indem Sie ganz einfach das Pluszeichen durch ein Minuszeichen ersetzen. Die Subtraktion setzt auch die Statusflags entsprechend dem Ergebnis der jeweiligen Operation, wobei das Carryflag in diesem Falle ein Borgen repräsentiert. Die Anweisung

```
SUB   AX,BX
```

subtrahiert den Wert im BX-Register vom Wert im AX-Register und läßt das Ergebnis im AX-Register. Die Statusflags werden entsprechend dem Ergebnis verändert.

Der Befehl SBB (Subtract with Borrow) löst das Problem der Subtraktion bei Zahlen von höherer Genauigkeit. SBB berücksichtigt nämlich das Carryflag als Teil der Subtraktion. In diesem Falle wird dann der Wert des Carryflags vom Ergebnis der normalen Subtraktion zusätzlich subtrahiert. Abbildung 4.10 zeigt eine Subtraktion mit Zahlen von höherer Genauigkeit, durchgeführt mit den gleichen Werten wie das Beispiel der Addition. Es werden also dann die Variablen VALUE1 und VALUE2 subtrahiert und das Ergebnis steht in VALUE2.

Arithmetik mit einem Operanden

Der Negationsbefehl NEG dient zum Vorzeichentausch. Der Befehl vertauscht dazu das Vorzeichen im Zweierkomplement eines Byte- oder Wortoperanden. Die beiden anderen Befehle, die nur einen Operanden benötigen, verändern den Wert des jeweiligen Operanden um 1. Der Befehl Increment (INC) addiert 1 auf den Operanden, während der Befehl Decrement (DEC) 1 vom jeweiligen Operanden subtrahiert. Increment und Decrement können beispielsweise dazu verwendet werden, mit einem Adresszeiger einen Speicherbereich schrittweise zu durchlaufen. Sie können auch als Schleifenzähler verwendet werden. Dabei erniedrigt jeder Durchlauf der Schleife den Zähler um 1. Wenn der Wert im Zähler dann 0 erreicht, ist die Schleife abgearbeitet.
Alle diese Befehle mit nur einem einzigen Operanden können sowohl einen Wort- als auch einen Byte-Speicherbereich als Operanden verwenden. Der Assembler benötigt allerdings zusätzliche Hilfe, wenn einer dieser Befehle einen Speicherbereich über indirekte Adressierung anspricht, wie z.B. [BX+SI]. Der Assembler muß nämlich den Typ des Speicheroperanden kennen, um den korrekten Operationscode festlegen zu können. Wir müssen dann die Angaben BYTE PTR oder WORD PTR machen, um den jeweiligen Operanden genau zu beschreiben.

Alle drei Befehle modifizieren das Statusregister auf die gleiche Art wie es die arithmetischen Anweisungen tun. Die Befehle "addiere 1", "subtrahiere 1" und "subtrahiere 1 von 0" entsprechen ganz genau den Befehlen INC, DEC und NEG. Die Befehle mit nur einem Operanden sind allerdings wirkungsvoller.

Vergleich

Der Vergleichsbefehl CMP vergleicht zwei Werte, indem er sie voneinander subtrahiert. CMP selbst liefert zwar kein Ergebnis, aber er setzt die Statusflags entsprechend dem Ergebnis. Der Befehl verändert also nur die Flags. Die Verwendung des Vergleichsbefehls geschieht dabei in der gleichen Weise wie wir den Subtraktionsbefehl verwenden. Allerdings gibt es keinen Vergleichsbefehl mit Übertrag.

Etwas aufwendiger als der Vergleich von Bytes und Wörtern ist der Vergleich von Zahlen mit höherer Genauigkeit. Und in der Tat ist es hier wesentlich einfacher, die Subtraktionsanweisung anzuwenden als den Vergleichsbefehl. Abbildung 4.11 zeigt den Vergleich eines Paares von 32-Bit Werten unter Benützung des AX-Registers als Zwischenspeicher. Dieser Vergleich stellt fest, welche der beiden Zahlen größer ist. Als Ergebnis setzen wir den entsprechenden Bedingungscode. Dabei bestimmt das Carryflag, welche der beiden Zahlen die größere ist. Enthält das Carryflag den Wert 1, so ist VALUE2 größer.

Das zweite Programm in Abbildung 4.11 testet zwei 32-Bit Zahlen auf Gleichheit. Halten wir dabei fest, daß unser Programm das Ergebnis des Vergleichs der niederwertigeren Zahlenteile speichert und es später mit dem Ergebnis des Vergleichs der höherwertigeren Zahlenteile kombiniert. Erst dann können wir feststellen, ob das Ergebnis wirklich 0 ist. Diesen Oder-Befehl werden wir zwar erst im nächsten

Abschnitt besprechen, doch wichtig ist hier nur zu wissen, daß er die beiden Werte in einer Art kombiniert, daß das Endergebnis nur dann 0 ist, wenn die beiden kombinierten Werte ebenfalls 0 sind. Das Nullflag übermittelt uns das Ergebnis dieser Vergleichs-Unterroutine. Ist das Nullbit 1, sind die beiden Zahlen gleich.

```
The IBM Personal Computer Assembler 01-01-83          PAGE     1-1
Figure 4.11 Multiprecision Compares

 1                                              PAGE    ,132
 2                                              TITLE   Figure 4.11 Multiprecision Compares
 3      0000                            CODE    SEGMENT
 4                                              ASSUME  CS:CODE,DS:CODE
 5
 6      0000  ????????                  VALUE1  DD      ?       ; 32 bit operand
 7      0004  ????????                  VALUE2  DD      ?
 8
 9      0008                            FIG4_11 PROC    NEAR
10
11                                      ;----- This subroutine compares two 32 bit numbers for unequal
12
13      0008                            COMPARE_UNEQUAL:
14      0008  A1 0000 R                         MOV     AX,WORD PTR VALUE1
15      000B  2B 06 0004 R                      SUB     AX,WORD PTR VALUE2      ; Subtract low order portion
16      000F  A1 0002 R                         MOV     AX,WORD PTR VALUE1+2
17      0012  1B 06 0006 R                      SBB     AX,WORD PTR VALUE2+2    ; Subtract the high 16 bits
18      0016  C3                                RET                             ; Return with flags set
19
20                                      ;----- This subroutine will compare two 32 bit numbers for equal
21
22      0017                            COMPARE_EQUAL:
23      0017  A1 0000 R                         MOV     AX,WORD PTR VALUE1
24      001A  2B 06 0004 R                      SUB     AX,WORD PTR VALUE2      ; Subtract low 16 bits
25      001E  8B D8                             MOV     BX,AX                   ; Save the low order result
26      0020  A1 0002 R                         MOV     AX,WORD PTR VALUE1+2
27      0023  2B 06 0006 R                      SUB     AX,WORD PTR VALUE2+2    ; Subtract the high 16 bits
28      0027  0B C1                             OR      AX,CX                   ; Combine the two numbers
29      0029  C3                                RET                             ; Zero flag indicates equal
30
31      002A                            FIG4_11 ENDP
32
33      002A                            CODE    ENDS
34                                              END
```

Abbildung 4.11 Vergleich mit höherer Genauigkeit

Dezimale Ausrichtung

Wir verwenden den gleichen Satz von arithmetischen Befehlen sowohl für das Rechnen im Zweierkomplement wie auch zur Behandlung von binär kodierten Dezimalzahlen (BCD). Allerdings kann das Ergebnis solcher Rechenoperationen für die Darstellung im BCD-Code falsch sein. Um dies nach dem Arbeiten mit der Arithmetik im Zweierkomplement wieder auszugleichen, verwenden wir die dezimalen Ausrichtungsbefehle.

Die Befehle DAA (Decimal Adjust for Addition) und DAS (Decimal Adjust for Subtraction) verwenden wir nur für Operationen mit gepackten BCD-Zahlen. Gepackte BCD-Zahl bedeutet, daß sich in einem Byte zwei Dezimalziffern befinden. Die Befehle DAA und DAS bearbeiten dabei nur jeweils ein Byte im AL-Register. Wegen dieser Einschränkung gibt es für die Befehle DAA und DAS auch keine Operanden.

Abbildung 4.12 zeigt zwei Beispiele. Im ersten Beispiel addieren wir zwei gepackte BCD-Zahlen. Beide BCD-Zahlen bestehen aus jeweils zwei Dezimalziffern, so daß sie in einem einzigen Byte dargestellt werden können. Wir addieren nun diese beiden Zahlen, das Ergebnis befindet sich dann im Register AL. Der direkt darauffolgende Befehl DAA korrigiert dieses Ergebnis und bringt es wieder in eine Form, die der gepackten BCD-Darstellung entspricht. Nach Ausführung des DAA-Befehls ist

der im Register AL enthaltene Wert wieder eine gültige gepackte BCD-Zahl mit einem Wert zwischen 0 und 99. War dieses Ergebnis kleiner als 100, so enthält das AL-Register den gültigen Wert, und das Carryflag ist 0. War das Ergebnis im Bereich zwischen 100 und 198, so enthält das AL-Register die beiden niedrigstwertigen Dezimalziffern und das Carryflag ist gesetzt, um anzuzeigen, daß ein Übertrag vorliegt.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.12 BCD Examples

1                                      PAGE    ,132
2                                      TITLE   Figure 4.12 BCD Examples
3      0000                    CODE    SEGMENT
4                                      ASSUME  CS:CODE,DS:CODE
5
6      0000  ??                BCD1    DB      ?       ; Two decimal digit BCD numbers
7      0001  ??                BCD2    DB      ?
8
9      0002  ????              BCD1L   DW      ?       ; Four decimal digit BCD numbers
10     0004  ????              BCD2L   DW      ?
11
12     0006                    FIG4_12 PROC    NEAR
13
14                             ;----- Add two packed BCD numbers
15
16     0006                    DAA_EXAMPLE:
17     0006  A0 0000 R                 MOV     AL,BCD1                 ; Get packed decimal number
18     0009  02 06 0001 R              ADD     AL,BCD2                 ; Add the second number
19     000D  27                        DAA                             ; Correct the result to BCD
20     000E  C3                        RET
21
22                             ;----- Add two 4 digit packed BCD numbers
23
24     000F                    DAA_LONG:
25     000F  A0 0002 R                 MOV     AL,BYTE PTR BCD1L
26     0012  02 06 0004 R              ADD     AL,BYTE PTR BCD2L       ; Add the low order BCD number
27     0016  27                        DAA                             ; Convert to BCD
28     0017  A2 0004 R                 MOV     BYTE PTR BCD2L,AL       ; Store the result
29     001A  A0 0003 R                 MOV     AL,BYTE PTR BCD1L+1
30     001D  02 06 0005 R              ADD     AL,BYTE PTR BCD2L+1     ; Add the high order BCD digits
31     0021  27                        DAA                             ; Correct the result
32     0022  A2 0004 R                 MOV     BYTE PTR BCD2L,AL       ; Store the result
33     0025  C3                        RET
34
35                             ;----- Subtract two packed BCD numbers
36
37     0026                    DAS_EXAMPLE:
38     0026  A0 0000 R                 MOV     AL,BCD1
39     0029  2A 06 0001 R              SUB     AL,BCD2                 ; Binary subtract of values
40     002D  2F                        DAS                             ; Convert following subtract
41     002E  C3                        RET
42
43     002F                    FIG4_12 ENDP
44
45     002F                    CODE    ENDS
46                                     END
```

Abbildung 4.12 BCD-Beispiele

Der DAA-Befehl setzt das Flagregister korrekt. Ist das Ergebnis einer Addition im Bereich zwischen 100 und 198, so zeigt das Carryflag an, daß ein Übertrag in die Hunderter-Position vorliegt. In ähnlicher Weise wird bei einem Nullergebnis das Nullflag gesetzt. Für Operationen mit gepackten BCD-Zahlen sind die Vorzeichen- und Überlaufflags ohne Bedeutung, obwohl das Vorzeichenflag gesetzt wird, wenn das höchstwertige Bit im AL-Register gesetzt ist. Der DAA-Befehl benützt nämlich ganz speziell das AUX-Flag, um festzustellen, ob er eine Korrektur vornehmen muß. Nach Ausführung des Befehls DAA ist der Inhalt des AUX-Flags unbestimmt.

Das zweite Beispiel in 4.12 zeigt eine BCD-Addition von höherer Genauigkeit. Dieser Vorgang ist vergleichbar mit binärer Arithmetik von höherer Genauigkeit, mit der Ausnahme, daß auf jede Byte-Addition ein DAA-Befehl folgt. Wegen der für den DAA-Befehl gültigen Beschränkungen können wir in diesem Beispiel die beiden gepackten Dezimalwörter nicht als Wörter bearbeiten und dann die Korrektur darauf anwenden. Auf gepackte BCD-Variablen darf nämlich nur Byte-Arithmetik angewandt werden.

In Abbildung 4.12 sehen wir schließlich noch die Anwendung des Befehls DAS. Auch hier verhält es sich ähnlich der binären Subtraktion, abgesehen davon, daß der DAS-Befehl der Subtraktion folgen muß. Auch hier sind wiederum nur Byte-Operationen zulässig.

ASCII-Ausrichtung: Addition und Subtraktion

Die ASCII-Ausrichtungsbefehle ähneln sehr den dezimalen Ausrichtungsbefehlen. Sie folgen den Additions- oder Subtraktionsoperationen bei ungepackten BCD-Zahlen. Werden in einem Programm die Anweisungen DAA und DAS bei gepackten BCD-Zahlen verwendet, so werden die entsprechenden ASCII-Ausrichtungsbefehle für ungepackte BCD-Zahlen angewandt. Bei ungepackten BCD-Zahlen repräsentiert ein einzelnes Byte jeweils eine Dezimalziffer zwischen 0 und 9. Dieses Zahlensystem wird auch als ASCII-Dezimal bezeichnet, da die Zahlen in einem Programm sehr einfach aus und nach der ASCII-Darstellung übernommen werden können (durch Addition oder Subtraktion von jeweils 30H).

Nach der Addition zweier ungepackter Dezimalzahlen benützen wir in einem Programm den Befehl AAA (ASCII Adjust for Addition), um das Ergebnis wieder in die korrekte ungepackte Darstellung zu überführen. Die Regeln für die Addition bleiben dabei die gleichen wie für gepackte Dezimalarithmetik. Da die Addition von zwei ungepackten Dezimalzahlen allerdings ein Ergebnis liefern kann, das größer als 9 ist, benötigen die Befehle AAA und AAS mehr Platz als das AL-Register. Beim Befehl AAA befindet sich beispielsweise nur die niederwertige Ziffer im Register AL. Trat während der Dezimaladdition ein Übertrag auf, so erhöht der Befehl den Inhalt des Registers AH um 1. Wenn das eintritt, werden vom Befehl AAA sowohl das Carryflag als auch das AUX-Flag auf jeweils 1 gesetzt, andernfalls auf 0. Alle anderen Flags sind nach Ausführung der Ausrichtungsanweisung unbestimmt. Die ASCII-Ausrichtungsanweisungen unterscheiden sich von ihren Dezimaläquivalenten dahingehend, daß sie im Falle eines Übertrags aus der niederwertigen Stelle sowohl das AH-Register als auch das Carryflag setzen.

Um ungepackte Dezimalzahlen voneinander zu subtrahieren, verwenden wir den Befehl AAS (ASCII Adjust for Subtraction), und zwar jeweils unmittelbar nach der Subtraktion. Dabei muß das Ergebnis der Subtraktion im AL-Register abgelegt sein. Das Ergebnis des ASCII-Ausrichtungsbefehls befindet sich wiederum im AL-Register. Trat bei der Subtraktion ein Übertrag auf, so erniedrigt die AAS-Anweisung den Inhalt des AH-Registers und setzt auch das Carry- und AUX-Flag. Andernfalls werden die Flags gelöscht. Alle übrigen Flags sind nach Ausführung dieses Befehls undefiniert.

Multiplikation

Der Mikroprozessor 8088 ist wesentlich leistungsfähiger als seine 8-Bit-Vorgänger. Eine der Gründe hierfür sind die zusätzlichen Multiplikations- und Divisionsanweisungen. Frühere Mikroprozessoren mußten auf Assemblerprogramme zurückgrei-

fen und Addition und Subtraktion verwenden, um Multiplikations- und Divisionsoperationen durchführen zu können.

Es gibt zwei Formen der Multiplikationsanweisung. Der Befehl MUL multipliziert zwei vorzeichenlose Zahlen und hat als Ergebnis ebenso eine vorzeichenlose Zahl. Der Befehl IMUL multipliziert zwei vorzeichenbehaftete Zahlen. Dabei werden die Zweierkomplemente der Zahlen als Operanden verwendet und das Ergebnis verfügt über korrekte Vorzeichen und Größe.

Beide Multiplikationsanweisungen können sowohl auf Byte- als auch auf Wortebene verwendet werden. Allerdings ist die Verwendung möglicher Operanden wesentlich mehr eingeschränkt als für Additions- und Subtraktionsbefehle. Abbildung 4.13 bietet einen Überblick über die möglichen Multiplikationsbefehle. Zur Multiplikation zweier 8-Bit Werte muß ein Operand sich immer im AL-Register befinden und das Ergebnis wird immer im AX-Register abgelegt. Das Ergebnis einer Multiplikation kann bis zu 16 Bits lang werden (der maximale Wert wäre dann $255 \times 255 = 65.025$). Für die Addition von zwei 16-Bit Zahlen muß ein Operand sich im AX-Register befinden. Das Ergebnis, das bis zu 32 Bits lang werden kann (der maximale Wert ist dann $65.535 \times 65.535 < 2^{32}$), befindet sich in einem Registerpaar. Dabei enthält das DX-Register die höherwertigen 16 Bits des Ergebnisses und das AX-Register die niederwertigen 16 Bits. Halten wir dabei fest, daß die Multiplikation keinen Direktwert als Operanden zuläßt.

Die Verwendung des Flagregisters bei der Multiplikation unterscheidet sich etwas von der Verwendung bei anderen arithmetischen Operationen. Die einzigen beiden gültigen Flags sind nämlich das Carry- und das Überlaufsflag. Und sie werden von den beiden Befehlen auch noch verschieden verwendet.

Die vorzeichenlose Multiplikation MUL setzt beide Flags, wenn die obere Hälfte des Ergebnisses ungleich 0 ist. Werden zwei Bytes miteinander multipliziert, so bedeutet das Setzen des Carry- und des Überlaufsflags, daß das Ergebnis größer als 255 ist und deshalb nicht in einem einzigen Byte Platz finden kann. Im Falle einer Wortmultiplikation werden die Flags gesetzt, wenn das Ergebnis größer als 65.535 wird.

Die Multiplikation vorzeichenbehafteter Integerzahlen IMUL setzt das Carry- und das Überlaufflag entsprechend den gleichen Kriterien — d.h. die Flags sind gesetzt, wenn das Ergebnis nicht in der unteren Hälfte des Ergebnisbereichs Platz findet. Allerdings ist hier der Vorgang, da die Zahlen mit Vorzeichen versehen sind, nicht einfach ein Vergleich der oberen Hälfte des Ergebnisses mit 0. Der Befehl IMUL setzt die Flags nur dann, wenn die obere Hälfte des Ergebnisses nicht der Vorzeichenerweiterung der unteren Hälfte des Ergebnisses entspricht. Dies bedeutet im Falle eines positiven Ergebnisses, daß der Test derselbe ist wie für den Befehl MUL — die Flags werden gesetzt, wenn die obere Hälfte des Ergebnisses ungleich 0 ist (das höchstwertige Bit ist jedoch 0 und zeigt damit ein positives Ergebnis an). Ist das Ergebnis negativ, so setzt der Befehl IMUL die Flags dann, wenn die obere Hälfte des Ergebnisses nicht aus lauter Einsen besteht (das höchstwertige Bit muß jedoch 1 sein und zeigt damit an, daß das Ergebnis negativ ist). So setzt beispielsweise eine Byte-Multiplikation mit negativem Ergebnis die Flags dann, wenn das Ergebnis kleiner als −128 ist, denn dies ist die kleinste darstellbare Zahl mit Vorzeichen inner-

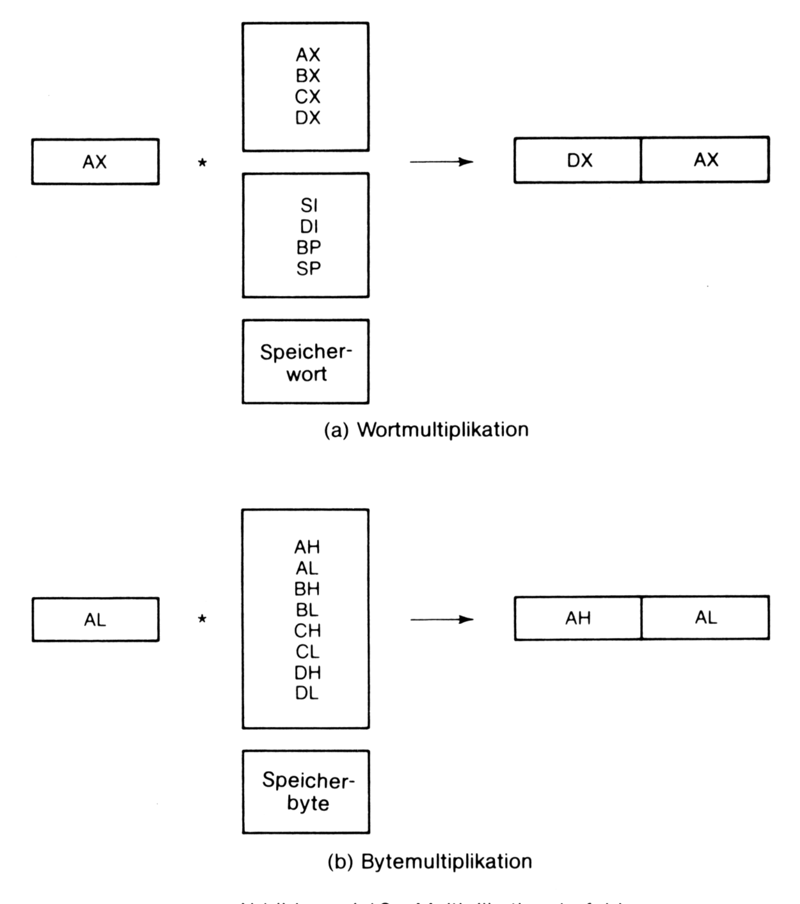


Abbildung 4.13 Multiplikationsbefehle

halb eines Bytes. Bei einer Wortmultiplikation mit positivem Ergebnis dagegen werden die Flags gesetzt, wenn das Ergebnis größer ist als 32.767, die größte Zahl, die in einem Wort mit Vorzeichen dargestellt werden kann.

ASCII-Ausrichtung: Multiplikation

Werden in einem Programm zwei ungepackte Dezimalzahlen multipliziert, so ist das Ergebnis im AL-Register eine Binärzahl. Nachdem die größte ungepackte BCD-Zahl 9 ist, ist das maximale Ergebnis einer ungepackten BCD-Multiplikation 81. Allerdings ist dieses Ergebnis keine gültige Darstellung einer ungepackten BCD-Zahl. Der Befehl AAM (ASCII Adjust for Multiply) konvertiert nun dieses binäre Ergebnis in eine ungepackte Dezimalzahl. Dies geschieht so, daß der Befehl AAM die höherwertige Dezimalziffer im AH-Register ablegt und die niederwertige im AL-Register. Multipliziert beispielsweise ein Programm die beiden Werte 6 und 7, so ist

das Ergebnis im AL-Register 2AH, wobei der folgende AAM-Befehl das Resultat dahingehend konvertiert, daß sich im AH-Register der Wert 04H befindet und im AL-Register der Wert 02H — oder die ungepackte Dezimalzahl 42 im Registerpaar AH:AL.

Der Befehl AAM erzeugt dieses ungepackte dezimale Ergebnis, indem er den im Register AL enthaltenen Wert durch 10 dividiert. Der Quotient wird dann in das AH-Register übertragen, der Rest der Division bleibt im AL-Register stehen. AAM setzt das Vorzeichen- und Nullflag in Entsprechung zum Ergebnis im AL-Register. Da dieses Ergebnis ungepackt dezimal ist, ist das Vorzeichen immer positiv, und auch das Nullflag wird nur dann gesetzt, wenn die Originalzahl ein Vielfaches von 10 ist — d.h. die niedrigstwertige Dezimalziffer ist 0. Alle anderen Flags werden vom Befehl unberührt gelassen. Das Carryflag spielt hier keine Rolle, da bei der Multiplikation von zwei ungepackten Dezimalzahlen niemals ein Ergebnis entstehen kann, das die Möglichkeiten der zweiziffrigen Dezimaldarstellung überschreiten würde.

Wir können den Befehl AAM auch dazu verwenden, um gegebenenfalls eine im Register AL enthaltene Binärzahl durch 10 zu dividieren. In diesem Fall müßten wir den Befehl als eine besondere Form der Division betrachten, bei der der im AL-Register enthaltene 1 Byte lange Wert durch 10 dividiert wird. Der Quotient befindet sich dann im AH-Register, der Divisionsrest bleibt im AL-Register.

Division

Der Befehlssatz des 8088 umfaßt als Teil der arithmetischen Funktionen auch die Division. Ähnlich der Multiplikation gibt es auch zwei Formen der Division — eine für vorzeichenlose Binärzahlen (DIV) und eine für vorzeichenbehaftete Zahlen im Zweierkomplement (IDIV). Jeder der beiden Befehle kann auf Byte- wie auch auf Wortoperanden angewandt werden.

Der DIV-Befehl führt eine vorzeichenlose Division des Dividenden aus und erzeugt sowohl Quotient als auch Divisionsrest. Ähnlich der Multiplikation müssen auch hier die Operanden an bestimmten Stellen abgelegt sein. Und ebenso weist einer der beteiligten Werte auch die doppelte Größe eines normalen Operanden auf. Im Falle der Division ist der Dividend der doppeltlange Operand. Der Byte-Befehl teilt einen 16-Bit Dividenden durch einen 8-Bit Divisor. Die Division erzeugt dabei zwei Ergebnisse. Der Quotient wird im AL-Register abgelegt, der Divisionsrest im AH-Register. Diese Anordnung der Operanden macht die Division zum direkten Gegenstück der Multiplikation. Die Multiplikation des AL-Registers mit einem Byte-Operanden und die nachfolgende Division des AX-Registers durch denselben Operanden stellt den ursprünglichen Inhalt des AL-Registers wieder her. Das AH-Register wäre dann auf 0 gesetzt, da es keinen Divisionsrest gibt. Abbildung 4.14 zeigt schematisch den Ablauf des Divisonsbefehls.

Der Wortbefehl dividiert einen 32-Bit Dividenden durch einen 16-Bit Divisor. Der Dividend befindet sich dabei im Registerpaar DX:AX, wobei DX den höherwertigen und AX den niederwertigen Teil enthält. Der Befehl legt den Quotienten dann im AX- und den Divisionsrest im DX-Register ab. Auch hier entsprechen sich Multiplikation

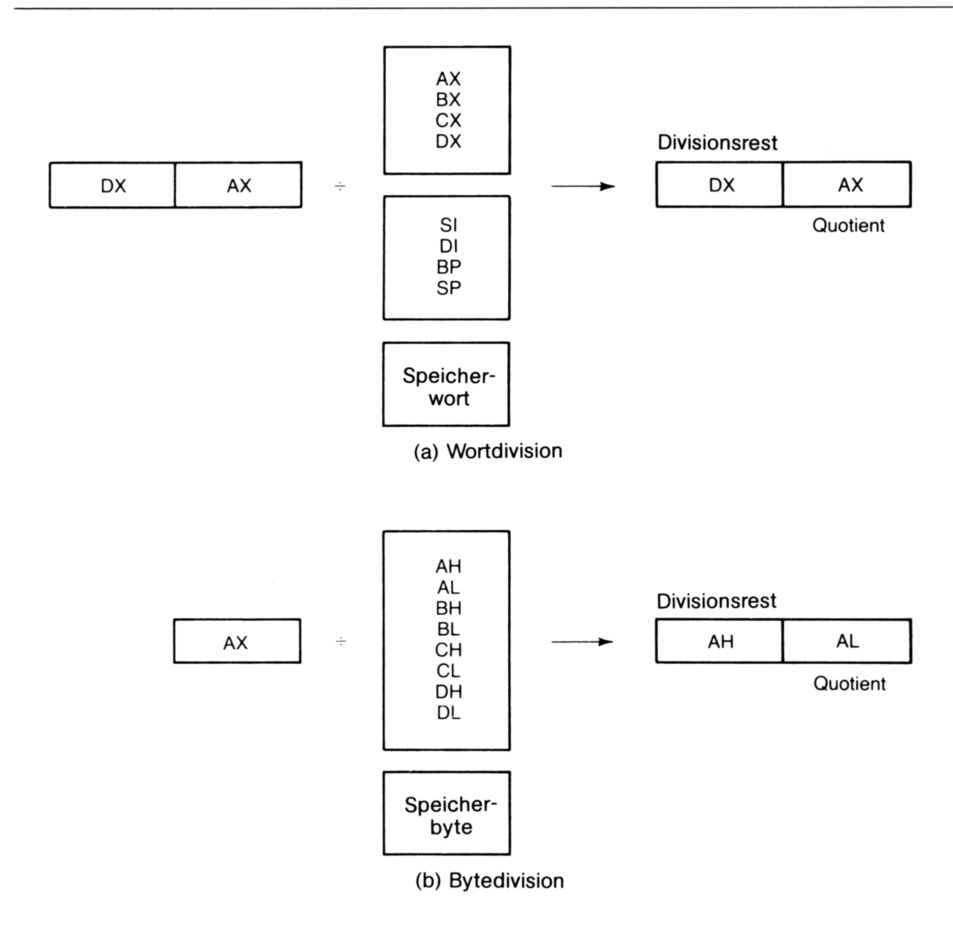


Abbildung 4.14 Divisionsbefehle

und Division wieder. Die Multiplikation von AX mit einem Wortwert und die darauffolgende Division durch denselben Wert stellt also in AX wieder den Ursprungszustand her. Das DX-Register wird auf Null gesetzt, da kein Divisionsrest entsteht.

Kein Statusflag wird durch einen Divisionbefehl irgendwie beeinflußt, jedoch kann während einer Division ein entscheidender Fehlerzustand auftreten. Ist der Quotient nämlich größer als das Ergebnisregister, so kann der Prozessor das korrekte Ergebnis nicht abspeichern. Für eine Byte-Division muß das Ergebnis deshalb kleiner als 256 und für eine Wort-Division kleiner als 65,536 sein. Der Divisionsbefehl selbst setzt keinerlei Flags um diesen Fehler anzuzeigen. Stattdessen führt der Prozessor eine Software-Unterbrechung, den sogenannten 0-Interrupt, aus. Wie alle Software-Interrupts rettet auch dieser Interrupt wegen Nulldivision die Flags sowie das CS- und IP-Register in den Stack. Der Prozessor überträgt die Steuerung sodann an die Stelle, die durch den Pointer bei Interrupt 0 bestimmt ist. Die Nulldivisionsroutine sollte nun geeignete Maßnahmen ergreifen, um die Fehlerbedingung zu behandeln.

(Interrupt 0 wird als Nulldivision bezeichnet, obwohl auch die Division durch eine andere Zahl als Null diese Unterbrechung auslösen kann. Auch in der vom Hersteller des 8088, Intel, veröffentlichten Literatur wird diese Unterbrechung als Nulldivision bezeichnet, obwohl wir sie eigentlich besser als Divisionsüberlauf bezeichnen müßten.)

Die Division vorzeichenbehafteter Zahlen (IDIV) unterscheidet sich vom DIV-Befehl eigentlich nur dadurch, daß sie die Vorzeichen der beiden Operanden berücksichtigt. Ist das Ergebnis positiv, läuft der Befehl wie die DIV-Anweisung ab, außer daß der maximale Wert des Quotienten 127 beim Byte-Befehl und 32,767 beim Wort-Befehl nicht übersteigen darf. Ist das Ergebnis jedoch negativ, so wird der Quotient gekürzt und der Divisionsrest erhält das Vorzeichen des Dividenden. Der kleinste Quotient für einen Byte-Befehl ist also −128 und für einen Wort-Befehl −32,768. Abbildung 4.15 zeigt vier Divisionsbeispiele mit den zugehörigen Ergebnissen.

Dividend (AX)	Divisor (mod-r/m)	Quotient (AL)	Divisionsrest (AH)
7	2	3	1
7	−2	−3	1
−7	2	−3	−1
−7	−2	3	−1

Abbildung 4.15 Division vorzeichenbehafteter Zahlen

Die gezeigten Beispiele sind Byte-Divisionen, wobei sich der Dividend im AX-Register befindet und der Divisor über das Mod-R/M-Byte bestimmt wird. Die Division legt den Quotienten dann im AL-Register und den Divisionsrest im AH-Register ab. Halten wir dabei fest, daß das Vorzeichen des Divisionsrestes immer dem des Dividenden entspricht. Der Quotient wird immer in Richtung auf Null verkürzt.

ASCII-Ausrichtung: Division

So wie bei allen anderen arithmetischen Befehlen gibt es auch für den Divisionsbefehl eine Variante, die die Verarbeitung ungepackter Dezimalzahlen in ASCII-Darstellung erlaubt. Im Gegensatz zum Vorgehen bei allen anderen Befehlen dieser Art muß die Anweisung AAD (ASCII Adjust for Division) jedoch dem eigentlichen Divisionsbefehl vorausgehen. Die AAD-Anweisung nimmt nämlich die zweiziffrige ungepackte Dezimalzahl aus dem AX-Register (höherwertiger Teil in AH), wandelt sie in eine Binärzahl um und legt diese im AL-Register ab, wobei das AH-Register auf Null gesetzt wird. Dadurch wird AX mit einem passenden Wert für die Division durch eine ungepackte einstellige Dezimalzahl belegt. Der AAD-Befehl setzt die Bedingungscodes entsprechend dem Ergebnis im AL-Register. Parity-, Vorzeichen- und Nullflag entsprechen dem Wert in AL, während alle anderen Flags undefiniert bleiben.

Nach Ausführung der Division muß der Quotient nicht unbedingt eine einstellige ungepackte Dezimalzahl sein. Dies ist in der Tatsache begründet, daß hier kein Divi-

sionsüberlauf erkannt wird. Der schlimmste Fall wäre die Division von 99 durch 1, die einen Quotienten von 99 erzeugen würde. Diese Zahl ist kleiner als das erlaubte Maximum sowohl für DIV als auch für IDIV, so daß kein Divisionsüberlauf entsteht. Allerdings ist diese Zahl größer als die größte ungepackte Dezimalzahl, nämlich 9. Es gibt nun zwei Wege, mit diesem Problem fertig zu werden. Einmal könnten wir auf jede Befehlsfolge AAD — DIV einen Test durchführen, um festzustellen, ob der Quotient größer ist als 9, und hierauf die geeignete Überlaufbehandlung vornehmen. Oder aber wir verwenden nach jeder Division den AAM-Befehl, um den Quotienten in eine zweistellige ungepackte Dezimalzahl zu überführen. Allerdings muß unser Programm dann vor Ausführung des AAM-Befehls den Divisionsrest irgendwohin sichern, denn dieser Befehl zerstört den Inhalt des AH-Registers. Auf diese Weise erhalten wir eine zweistellige Dezimalzahl als Ergebnis der Division einer zweistelligen durch eine einstellige Zahl. Ist allerdings die als Divisor verwendete ungepackte Dezimalzahl gleich Null, so wird diese Division den Interrupt für Nulldivision auslösen und so den Divisionsüberlauf anzeigen.

Konvertierung

Führt ein Programm Divisionen mit vorzeichenbehafteten ganzen Zahlen durch, so entsteht ein Problem, wenn der Dividend ein Byte-Operand ist. Es ist sicherlich in Ordnung, einen Byte-Wert durch einen Byte-Wert zu dividieren, doch der Divisionsbefehl verlangt den Dividenden im AX-Register. Für die Division unter Berücksichtigung des Vorzeichens muß AX nun das korrekte Zweierkomplement der jeweiligen Zahl enthalten. Der Befehl CBW (Convert Byte to Word) erfüllt diese Aufgabe, indem er den Wert im AL-Register in das AH-Register vorzeichenerweitert. Das bedeutet, daß AH mit Nullen gefüllt wird, wenn der Wert in AL positiv ist. Ist AL dagegen negativ, so wird AH mit Einsen gefüllt. Der CBW-Befehl belegt also das 16-Bit AX-Register mit dem gleichen Wert wie er ursprünglich im AL-Register enthalten war. Für Wort-Divisionen erfüllt der Befehl CWD (Convert Word to Doubleword) die gleiche Aufgabe. Der Wert in AX wird in diesem Fall nach DX vorzeichenerweitert. Die beiden Befehle dienen also zum Erweitern der Operanden vor dem Durchführen von Divisionen mit Vorzeichen.

Für die Division von vorzeichenlosen Zahlen ist unter denselben Umständen keine Vorzeichenerweiterung in die höherwertigen Teile des Dividenden notwendig. Es genügt in diesen Fällen ganz einfach, vor Durchführen der Division das AH- (oder DX-) Register mit Nullen zu füllen. Wir können dafür viele Anweisungen einschließlich eines MOV-Befehls mit Direktwert verwenden, oder aber auch ganz einfach den Befehl

```
SUB     AH,AH
```

benützen, der uns garantiert, daß der Inhalt von AH Null ist.

Rechenbeispiel

Um die Befehle, die wir in den vorausgehenden Abschnitten besprochen haben, etwas zu veranschaulichen, wollen wir eine kleine Rechenaufgabe in Assembler lösen. Unser Beispiel ist einfach, doch benützt es viele der besprochenen Anweisungen. Die Aufgabe besteht darin, den Quotienten aus zwei arithmetischen Ausdrücken zu berechnen, wobei einige der Werte Konstanten und einige Variablen sind. Alle Werte sind jedoch 16-Bit-Werte mit Vorzeichen.

Die Rechenformel lautet:

$$X = \frac{A \times 2 + B \times C}{D - 3}$$

Die Assembler-Routine in Abbildung 4.16 löst die gestellte Aufgabe. Als erstes werden dabei zwei Multiplikationen durchgeführt. Da der 8088 das Ergebnis einer 16-Bit Multiplikation immmer im DX:AX-Registerpaar ablegt, verschieben wir das Ergebnis vor Ausführen der zweiten Multiplikation in das BX:CX-Registerpaar. Nach diesen Multiplikationen werden nun die beiden Werte im Zähler addiert. Da die

```
The IBM Personal Computer Assembler 01-01-83         PAGE    1-1
Figure 4.16 Arithmetic Example

 1                                     PAGE    ,132
 2                                     TITLE   Figure 4.16 Arithmetic Example
 3
 4                              ;-----------------------------------------------
 5                              ; This example calculates the formula
 6                              ;
 7                              ;               A * 2  +  B * C
 8                              ;    X  =      ---------------
 9                              ;                  D  -  3
10                              ;
11                              ; All variables are 16 bit signed integers.
12                              ;-----------------------------------------------
13
14   0000                       CODE    SEGMENT
15                                      ASSUME  CS:CODE,DS:CODE
16
17   0000  ????                 X       DW      ?          ; Storage for variables
18   0002  ????                 A       DW      ?
19   0004  ????                 B       DW      ?
20   0006  ????                 C       DW      ?
21   0008  ????                 D       DW      ?
22
23   000A                       FIG4_16 PROC    NEAR
24
25   000A  B8 0002                      MOV     AX,2       ; Set up the constant
26   000D  F7 2E 0002 R                 IMUL    A          ; DX:AX = A * 2
27   0011  8B DA                        MOV     BX,DX
28   0013  8B C8                        MOV     CX,AX      ; BX:CX = A * 2
29   0015  A1 0004 R                    MOV     AX,B
30   0018  F7 2E 0006 R                 IMUL    C          ; DX:AX = B * C
31   001C  03 C1                        ADD     AX,CX
32   001E  13 D3                        ADC     DX,BX      ; DX:AX = A * 2 + B * C
33   0020  8B 0E 0008 R                 MOV     CX,D
34   0024  83 E9 03                     SUB     CX,3       ; CX = D - 3
35   0027  F7 F9                        IDIV    CX         ; AX = (A*2 + B*C)/(D-3) Quotient
36   0029  A3 0000 R                    MOV     X,AX       ; Store the result
37   002C  C3                           RET
38
39   002D                       FIG4_16 ENDP
40   002D                       CODE    ENDS
41                                      END
```

Abbildung 4.16 Rechenbeispiel

Multiplikationen 32-Bit-Resultate erzeugten, benötigen wir jetzt eine Addition mit größerer Genauigkeit. Diese Addition legt ihr Ergebnis im DX:AX-Registerpaar ab. Nun berechnen wir den Nenner im CX-Register und führen hierauf die Division Zähler : Nenner durch. Der in AX enthaltene Quotient wird sodann in die Variable X gespeichert. Unser Programm vernachlässigt also den Divisionsrest.

Logische Befehle

Die nächste Klasse von Befehlen sind die logischen Befehle. Diese Befehle verändern Daten ähnlich den arithmetischen Anweisungen, doch tun sie dies auf einer gänzlich anderen Basis. Während Additions- und Subtraktionsbefehle noch mit der Schulmathematik verwandt sind, arbeiten die logischen Befehle ausschließlich auf Basis der Einsen und Nullen, mit denen ein Computer arbeitet. Allgemein gesagt erlauben es diese Befehle einem Programm, Operationen auf Bit-Ebene durchzuführen.

Die vier Hauptfunktionen sind dabei AND, OR, XOR und NOT. Es gibt noch andere logische Funktionen, die aus diesen vier Grundbefehlen zusammengesetzt sind, doch verfügt der 8088 über keine eigenen Befehle hierfür. Die vier genannten Funktionen arbeiten gänzlich auf der Basis der binären Datendarstellung.

Der NOT-Befehl ist der einfachste. Er basiert auf der wahr/falsch-Definition von 0 und 1. NOT „wahr" ist dann „falsch", und NOT „falsch" ist „wahr". Der NOT-Befehl komplementiert also alle Bits des jeweiligen Datenwerts. Von einem anderen Standpunkt aus könnte man sagen, daß der NOT-Befehl einer Subtraktion der jeweiligen Daten von einem Wert aus lauter Einsen entspricht. Abbildung 4.17 zeigt die Wirkung des NOT-Befehls auf ein einzelnes Bit.

Wert	NOT (Wert)
0	1
1	0

Abbildung 4.17 NOT-Befehl

Die verbleibenden drei logischen Funktionen benötigen alle zwei Operanden. Abbildung 4.18 zeigt das Ergebnis der drei Funktionen an jeweils einem Paar Bits.

X	Y	X AND Y	X OR Y	X XOR Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Abbildung 4.18 Logische Befehle

Die Abbildung zeigt das Ergebnis eines jeden Befehls an jeweils einem einzigen Bitpaar. Da der 8088 mit Byte- bzw. Wortoperanden arbeitet, wiederholen sich die Ergebnisse entsprechend für jede einzelne Bitposition. So verknüpft beispielsweise der AND-Befehl Bit 0 der beiden Operanden, um dann das Ergebnis der Operation wiederum in Bit 0 des Zieloperanden abzulegen. Dieser Vorgang wiederholt sich für die Bits 1 bis 7. Das Ergebnis ist die bitweise logische Verknüpfung der beiden Operanden.

Das Ergebnis des AND-Befehls ist nur dann 1, wenn beide Operanden auf 1 stehen. In Wahr-/Falsch-Terminologie ausgedrückt ist das Ergebnis nur dann wahr, wenn die beiden Operanden X und Y wahr sind. Die OR-Funktion liefert 1 als Ergebnis, wenn einer der beiden Operanden oder aber beide Operanden 1 sind. Das Ergebnis ist also wahr, wenn X oder Y wahr ist. Diese Oder-Operation wird manchmal auch „Inklusives Oder" genannt, um sie von dem anderen Operator, dem „Exklusiven Oder" zu unterscheiden. Das Ergebnis von X XOR Y ist nur dann 1, wenn einer der Operanden 1 und der andere 0 ist. Sind beide Operanden 0 oder sind beide Operanden 1, so ist das Ergebnis 0. Das Exklusiv-Oder funktioniert also wie eine Addition ohne Übertrag.

Abbildung 4.19 zeigt die logischen Befehle des 8088. Der NOT-Befehl mit einem einzigen Operanden hat eine Form, die praktisch dem der Negationsanweisung (NEG) entspricht. Die restlichen logischen Befehle entsprechen in ihrer Syntax der Addition und Subtraktion.

Auch wenn der 8088 eine logische Operation durchführt, setzt er die Flags in Entsprechung zum Ergebnis. Da es sich hier nicht um arithmetische Operationen handelt, sind das Carry- und Überlaufsflag immer auf 0 gesetzt. Das AUX-Flag bleibt bei logischen Operationen undefiniert, während die restlichen Flags (Vorzeichen-, Null- und Parityflag) korrekt das Ergebnis der jeweiligen Operation wiederspiegeln. Die Ausnahme ist der NOT-Befehl, der die Flags überhaupt nicht verändert.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.19 Logical Instructions

1                                              PAGE    ,132
2                                              TITLE   Figure 4.19 Logical Instructions
3       0000                           CODE    SEGMENT
4                                              ASSUME  CS:CODE,DS:CODE
5       0000                           EXBYTE  LABEL   BYTE
6       0000                           EXWORD  LABEL   WORD
7
8       0000  22 06 0000 R                     AND     AL,EXBYTE               ; AL <- AL AND [EXBYTE]
9       0004  81 E3 9FEF                       AND     BX,1001111111101111B    ; BX <- BX AND 9FEFH
10      0008  80 26 0000 R 03                  AND     EXBYTE,00000011B        ; [EXBYTE] <- [EXBYTE] AND 3
11
12      000D  08 2E 0000 R                     OR      EXBYTE,CH               ; [EXBYTE] <- [EXBYTE] OR CH
13      0011  0B 16 0000 R                     OR      DX,EXWORD               ; DX <- DX OR [EXWORD]
14      0015  0D FFF9                          OR      AX,0FFF9H               ; AX <- AX OR 0FFF9H
15
16      0018  33 1E 0000 R                     XOR     BX,EXWORD               ; BX <- BX XOR [EXWORD[
17      001C  30 1E 0000 R                     XOR     EXBYTE,BL               ; [EXBYTE] <- [EXBYTE] XOR BL
18      0020  34 EF                            XOR     AL,0EFH                 ; AL <- AL XOR 0EFH
19
20      0022  F7 D1                            NOT     CX                      ; CX <- NOT CX
21      0024  F6 16 0000 R                     NOT     EXBYTE                  ; [EXBYTE] <- NOT [EXBYTE]
22
23      0028  F7 06 0000 R 0003                TEST    EXWORD,0003H            ; Set flags for [EXWORD] AND 3
24      002E  84 E0                            TEST    AH,AL                   ; Set flags for AH AND AL
25      0030  A9 0002                          TEST    AX,02H                  ; Set flags for AX AND 2
26
27      0033  D1 C1                            ROL     CX,1                    ; Left rotate one position
28      0035  D3 0E 0000 R                     ROR     EXWORD,CL               ; Right rotate variable amount
29
30      0039  D0 16 0000 R                     RCL     EXBYTE,1                ; Left rotate one position
31      003D  D3 DB                            RCR     BX,CL                   ; Right rotate variable amount
32
33      003F  D1 E0                            SHL     AX,1                    ; Left shift one position
34      0041  D1 E0                            SAL     AX,1                    ; Identical instruction
35
36      0043  D3 EB                            SHR     BX,CL                   ; Right shift variable amount
37      0045  D0 3E 0000 R                     SAR     EXBYTE,1                ; Right arithmetic shift
38
39      0049                           CODE    ENDS
40                                             END
```

Abbildung 4.19 Logische Befehle

Das Hauptanwendungsgebiet der logischen Befehle beim 8088 ist die Bitmanipulation. Die kleinste Dateneinheit, die der 8088 bearbeiten kann, ist nämlich das Byte. Kein einziger arithmetischer Befehl kann ein einzelnes Bit direkt ansprechen. Die logischen Befehle erlauben es unserem Programm nun, im Gegensatz dazu mit einzelnen Bits umzugehen.

Wozu dienen nun Operationen mit einzelnen Bits? Oft kommt es vor, daß ein Programm Werte speichern soll, die nur aus einer Anzeige „wahr" oder „falsch" bestehen. Ein Bit kann z.B. genügen, um anzuzeigen, daß der Drucker gerade tätig, eine Taste gedrückt oder die Initialisierung eines Programmteils abgeschlossen ist. In solchen Fällen würde es Verschwendung von Computerspeicher bedeuten, wenn wir jedesmal ein 8 Bit langes Byte verwenden müßten, um diese einzelne Information zu speichern. Wir können nun eine Anzahl solcher einzelner Informationsbits jeweils in einem Byte kombinieren, wenn wir über eine Methode verfügen, einzelne Bits zum Testen und Setzen isolieren zu können. Diese Kombination einzelner Bit-Flags wird hauptsächlich in Ein-/Ausgabe-Geräten verwendet, deren Hardware auf verschiedene Adressen ansprechen kann. Es ist für diese Geräte nämlich wesentlich einfacher, an einer einzigen Adresse mehrere Bits zu entschlüsseln, als jeweils eine eigene Adresse verwenden zu müssen.

Logische Befehle dienen nun dazu, einzelne Bits in einem Byte oder Wort zu isolieren, so daß wir sie setzen, löschen oder testen können. Zu diesem Zweck verwenden wir Maskenwerte, um die einzelnen Bits zu isolieren. Der Befehl wendet diesen Maskenwert Bit für Bit auf den entsprechenden Datenwert an. Um ein einzelnes Bit zu setzen, verwenden wir den Oder-Befehl. Die Maske enthält dann lauter Nullen, mit Ausnahme desjenigen Bits, das wir setzen wollen. Die Oder-Verknüpfung des Maskenwertes und des Operanden setzt dann eine 1 an das ausgewählte Bit und läßt alle anderen Bits unverändert. Auf ähnliche Weise kann der AND-Operator ein einzelnes Bit löschen. In diesem Falle setzen wir die Maske auf lauter Einsen mit Ausnahme des Bits, das wir löschen wollen. Diese Position wird dann immer 0 annehmen, während alle anderen Bits unverändert bleiben.

Normalerweise verwenden wir das exklusive Oder (XOR) nicht so häufig wie den AND- oder OR-Befehl, doch auch dieser Befehl hat einen Sinn für uns. Wir können XOR nämlich dazu verwenden, ein einzelnes Bit zu komplementieren. Dazu setzen wir die Maske des Befehls XOR so, daß das zu komplementierende Bit eine 1 erhält und alle anderen Bits auf 0 stehen. Wird der Befehl XOR nun ausgeführt, so bleiben alle Bits, die den auf 0 gesetzten Maskenbits entsprechen, unverändert. Die restlichen Bits werden komplementiert. Ist der Originalwert dabei 0, so ergibt 1 XOR 0 den Wert 1, also das Komplement von 0. Ist der Originalwert 1, so ergibt 1 XOR 1 den Wert 0, also das Komplement von 1. Die drei logischen Operatoren erlauben es unserem Programm also, einzelne Bits in einem Datenbereich zu setzen, zu löschen oder zu komplementieren.

Der letzte logische Befehl ist schließlich die Testanweisung. Dieser Befehl entspricht dem AND-Befehl, abgesehen davon, daß er das Zielfeld nicht verändert. Er setzt nur die Flag-Werte entsprechend dem Ergebnis. Das heißt, der Befehl TEST verhält sich zum Befehl AND so wie der Befehl CMP zum Befehl SUB. Er testet dabei ein bestimmtes Bit oder eine Anzahl von Bits innerhalb eines Bytes oder Worts.

Wie funktioniert nun dieser Test? Nehmen wir einmal an, unser Programm will das niederwertigste Bit eines Bytes, also Bit 0, testen. Zu diesem Zweck verwenden wir eine Maske mit dem Wert 01H entweder in einem Register oder als Direktwert. Der Befehl TEST (oder AND) erzeugt nun ein Ergebnis, das garantiert an allen Stellen 0 enthält, mit Ausnahme des Bits 0. Der Inhalt des Bits 0 reflektiert dabei den jeweiligen Originalwert. Da das Originalbit 0 ist, bleibt auch Bit 0 auf Null. War das Originalbit 1, so wird Bit 0 zu 1. Wichtiger noch, das Ergebnis dieses Befehls, wie es sich im Statusflag wiederspiegelt, zeigt auch den Status dieses Bits an. Enthält das getestete Bit eine Eins, so ist das Ergebnis nicht Null, und das Nullflag ist gelöscht. Enthält das zu testende Bit eine Null, so ist auch das Ergebnis Null, und das Nullflag ist gesetzt. Wir können also auf diese Weise mit den Befehlen TEST oder AND mittels einer Maske den Wert eines einzelnen Bits überprüfen, wobei diese Maske nur an der Stelle des zu überprüfenden Bits eine Eins enthalten muß. Das Flagregister zeigt dann den Zustand dieses zu prüfenden Bits. Der TEST-Befehl überprüft dabei den Inhalt der einzelnen Bits ohne die Information in den anderen Bits zu zerstören, da der Inhalt des Zieloperanden nicht verändert wird.

Schiebe- und Rotationsbefehle

Die verbleibenden logischen Befehle in Abbildung 4.19 dienen zum Schieben von Daten. Eine solche Schiebeoperation verschiebt dabei alle Bits in einem Datenbereich entweder nach links oder nach rechts. Stellen wir uns dazu eine mit Männlein und Weiblein besetzte Kirchenbank vor. Kommt nun eine neue Person in der Kirche

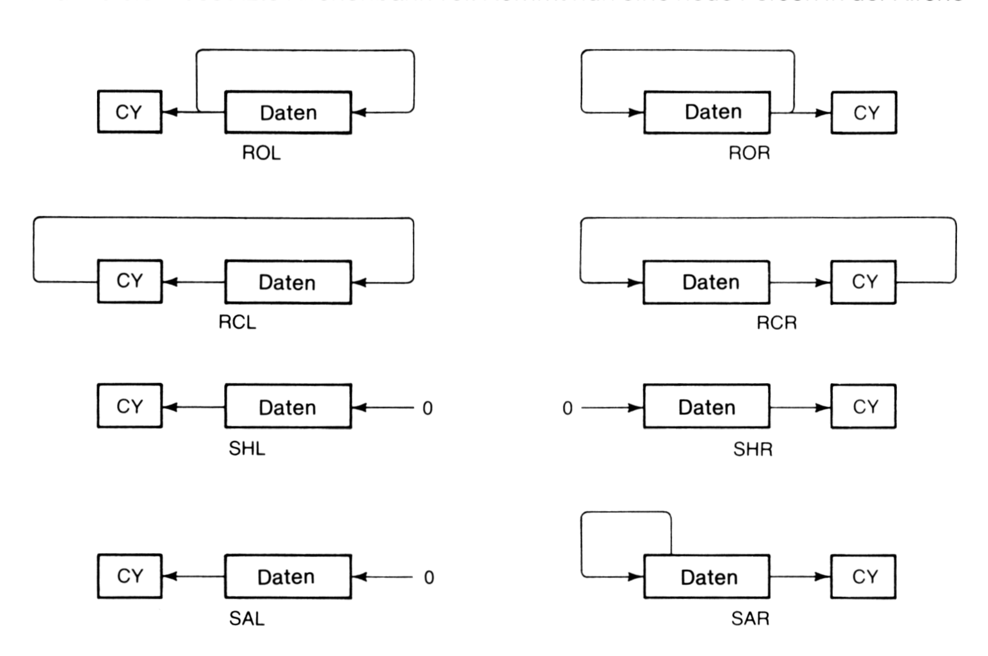


Abbildung 4.20 Schiebebefehle

an und setzt sich in die Bank, so müssen alle darin befindlichen Personen um eine Position zur Seite rücken. Ist die Bank jedoch voll, so fällt bei diesem Aufrücken um jeweils eine Person die letzte Person aus der Bank. Ein Schiebebefehl führt nun genau das gleiche durch, wobei die Männer und Frauen in der Kirchenbank hier durch Einsen und Nullen ersetzt werden.

Abbildung 4.20 zeigt die acht verfügbaren Schiebe- und Rotationsbefehle. Auch innerhalb dieser Befehle gibt es noch Variationen. Zuerst sehen wir uns einmal an, was diese Befehle gemeinsam haben.

Wie auch die anderen logischen Befehle arbeiten die Schiebe- und Rotationsbefehle auf Byte- oder Wortbasis. Jeder Befehl benötigt dabei nur einen Operanden. Dieser Operand kann entweder ein Register oder eine Speicherstelle sein. Alle Befehle benützen das Mod-R/M-Byte, um diesen Operanden zu spezifizieren.

Alle Shift- und Rotationsbefehle benötigen einen speziellen Schiebezähler. Das bedeutet, das Programm muß die Anzahl der Bits, um die jeweils geschoben werden soll, festlegen. Dieser Wert ist dann der Schiebezähler. Der am meisten vorkommende Wert hierfür ist Eins. Dies bedeutet, daß die Bits in dem jeweiligen Operanden um eine Stelle verschoben werden. Übrigens können wir bei jedem Befehl eine beliebige Anzahl von Bits angeben, um die geschoben werden soll, indem wir den jeweiligen Shift-Zähler im CL-Register speichern, bevor wir den Befehl ausführen. Geben wir nun bei der Ausführung des Befehls an, daß sich der Shift-Zähler im CL-Register befindet, dann bestimmt dieser Zähler die Anzahl der Bits, um die geschoben werden soll. Der Wert im CL-Register darf dabei jede Zahl zwischen 0 und 255 sein, doch im allgemeinen werden nur Werte zwischen 0 und 16 verwendet. Ein Wert von 0 bewirkt dabei, daß überhaupt nicht verschoben wird, während jeder Wert größer als 16 mehr Bits verschiebt als der Operand überhaupt enthält.

Ein gemeinsamer Punkt für alle Schiebebefehle ist das Setzen des Carryflags. Das Bit, das am Ende des Datenbereichs jeweils wegfällt, hat eine besondere Stellung. Die Schiebe- und Rotationsbefehle stellen nämlich den letzten aus dem Operanden geschobenen Wert in das Carryflag. Wurde dabei nur um ein einziges Bit geschoben, so wird das Bit am äußeren Ende des Datenbereichs zum neuen Wert des Carryflags. Bei einem Schiebebefehl über mehrere Bits kommt das schließlich im Carryflag enthaltene Bit aus einer Stelle innerhalb des Operanden. Das Carryflag ist bei Operationen von höherer Genauigkeit von Bedeutung. Da ein Shift-Operand maximal 16 Bits lang sein kann, müssen wir bei größeren Datenbereichen mehrfache Schiebeoperationen und das Carryflag verwenden. Wir brechen dazu den Operanden in einzelne 16-Bit Teile auf und verschieben dann jeden einzelnen Teil um jeweils 1 Bit. Wir können jetzt das Carryflag in unserem Programm verwenden, um die beim vorausgegangenen Befehl hinausgeschobene Information beim Schieben des nächsten Teils wieder zu verwenden.

Die ersten vier Anweisungen in Abbildung 4.20 sind Rotationsbefehle. Das Bild enthält dazu eine schematische Darstellung des Ablaufs eines jeden Befehls. Die Rotationsbefehle nehmen, wie das Bild zeigt, das am Ende eines Datenbereichs herausfallende Bit und fügen es am Anfang des Datenbereichs wieder ein. Die Befehle „Rotieren links" (Rotate left — ROL) und „Rotieren rechts" (Rotate right — ROR) unter-

scheiden sich nur durch die Richtung des jeweiligen Datentransports. Ebenso spiegelbildlich verhalten sich die Befehle RCL (Rotate Left with Carry) und RCR (Rotate Right with Carry). Die Befehle ROL und RCL unterscheiden sich dabei in der Behandlung des Carryflags. Ein ROL-Befehl auf Byte-Basis rotiert 8 Datenbits entsprechend dem Schiebezähler. Ein auf Byte-Basis ausgeführter RCL-Befehl behandelt dagegen die Daten in einer Breite von 9 Bits, wobei das Carryflag das neunte Bit darstellt. Ein ROL-Befehl auf Wortbasis rotiert entsprechend 16 Bits und ein RCL-Befehl auf Wortbasis entsprechend 17 Bits.

Die Schiebebefehle am Ende von Abbildung 4.20 bringen die hinausgeschobenen Bits nicht mehr in den Operanden zurück. Sie werden über das Carryflag ins Nichts geschoben. Dabei bestimmt der Typ der Schiebeoperation außerdem, welcher Wert in den Operanden nachgeschoben wird. Für einen logischen Shift-Befehl wird der im Operanden frei werdende Platz immer mit Nullen aufgefüllt. Ein arithmetischer Schiebebefehl füllt dagegen den freien Datenbereich in Abhängigkeit vom Vorzeichen mit Nullen oder Einsen auf.

Warum nennen wir nun einen Schiebebefehl „arithmetischen Schiebebefehl", wenn er innerhalb der logischen Befehle auftaucht? Ganz einfach, das Schieben einer Zahl um eine Bit-Position entspricht nämlich einer Multiplikation oder Division dieser Zahl mit 2. Im Dezimalsystem würde das Hinzufügen einer Null an der letzten Stelle bedeuten, die Zahl mit 10 zu multiplizieren. Im Binärsystem bedeutet das Hinzufügen einer Null an der letzten Stelle eine Multiplikation mit 2. Da der Rechner nicht einfach ein Bit an das Ende einer Zahl anfügen kann, erfüllt eine Schiebeoperation denselben Zweck. Ein Schiebebefehl nach links schiebt alle Bits um eine Position nach links und fügt eine Null an der frei gewordenen niedrigstwertigen Bitposition ein. Auf diese Weise entspricht ein Schiebebefehl um 1 nach links einer Multiplikation der Zahl mit 2. Ist der Schiebezähler größer als 1, so entspricht die Zahl, mit der wir multiplizieren, der Zahl 2 in der entsprechenden Potenz. So entspricht beispielsweise ein Schieben um 3 Bits nach links einer Multiplikation mit 2^3, also 8.

Das Schieben einer Zahl um eine Stelle nach rechts stellt entsprechend eine Division durch 2 dar. Der zu schiebende Operand ist dabei der Quotient und das Carryflag der Divisionsrest. Ist der Schiebezähler größer als 1, bleibt der Operand immer noch der Quotient, der Divisionsrest jedoch entfällt. Solchermaßen stellen Schiebeoperationen eine sehr effektive Methode der Multiplikation oder Division mit einer Potenz von 2 dar. Und in der Tat kann diese Fähigkeit sogar dann ein Ersatz für eine Multiplikation sein, wenn der Multiplikand nicht eine Potenz von 2 ist.

Probleme mit den arithmetischen Schiebebefehlen treten dann auf, wenn wir eine negative Zahl nach rechts verschieben, um sie durch 2 zu dividieren. Füllt der Schiebebefehl nämlich die höchste Bitposition mit 0 auf, so wird das Ergebnis positiv und der absolute Wert der Zahl die Hälfte des Originalwertes. Der Befehl SAR (Shift Arithmetic Right) löst dieses Problem, indem er das höchstwertige Bit unverändert läßt, die anderen Bits aber verschiebt. Auf diese Weise bleibt eine negative Zahl negativ und eine positive positiv. Das Problem tritt beim Schieben nach links nicht auf, da sich das Vorzeichenbit am linken Ende des Operanden befindet. Des-

halb sind die Befehle „Shift Logical Left" (SLL) und „Shift Arithmetic Left" (SAL) im Grunde genommen identisch.

Aufgrund der arithmetischen Natur der Schiebe- und Rotationsbefehle wird von ihnen allen sowohl das Überlaufs- als auch das Carryflag bedient. Dabei ist das Überlaufsflag bei Schiebebefehlen mit mehr als einem Bit undefiniert. Bei Schiebebefehlen um jeweils ein Bit allerdings wird von den Schiebe- bzw. Rotationsbefehlen das Überlaufsflag gesetzt, falls sich das Vorzeichen der Zahl im Ergebnis verändert. Hat sich das höchstwertige Bit nicht verändert, wird das Überlaufsbit auf 0 gesetzt. Dadurch können wir über das Überlaufsflag feststellen, ob das implizite Multiplizieren oder Dividieren mittels einer Schiebe- oder Rotationsanweisung als Ergebnis ein gültiges Zweierkomplement erzeugt hat.

Zwei weitere Beispiele von Schiebe- bzw. Rotationsbefehlen sehen wir in Abbildung 4.21. Das erste Beispiel zeigt, wie ein Wert mittels der SAL-Anweisung multipliziert wird. Dabei multiplizieren wir mit 9, also keiner Potenz von 2. Im Beispiel werden dabei die Daten um drei Positionen nach links verschoben, um sie mit 8 zu multiplizieren. Im weiteren wird dann dieses Ergebnis auf den Originalwert addiert und so ein Resultat erreicht, das einer Multiplikation des Originalwertes mit 9 entspricht.

Die Nachteile dieser Methode sind augenscheinlich. Sie benötigt wesentlich mehr Befehle als eine einfache Multiplikation, die etwa so aussehen würde:

```
PUSH   DX
MOV    DX,9
IMUL   DX
POP    DX
```

Auch erzeugt unsere Schiebemultiplikation mit 9 ein 16-Bit Ergebnis anstelle des 32-Bit Ergebnisses des Befehls IMUL.

Dennoch kann die Multiplikation mittels Schiebebefehlen in einigen Fällen wünschenswert erscheinen. Ihr Hauptvorteil ist die hohe Verarbeitungsgeschwindigkeit. Der Befehl IMUL benötigt einen sehr großen Zeitaufwand, wogegen die Schiebebefehle sehr schnell ablaufen. Bei dem Beispiel in Abbildung 4.21 läuft die Schiebemethode etwa 25% schneller ab. Das ist zwar kein enormer Zeitgewinn, könnte aber entscheidend sein, wenn eine bestimmte Anwendung ganz auf der Multiplikation mit 9 aufgebaut ist. Die Multiplikation mit einer Potenz von 2 würde dabei noch erheblich größere Zeitgewinne ermöglichen.

Das zweite Beispiel in Abbildung 4.21 zeigt die Anwendung des Schiebezählers, um damit eine Bitselektion zu erreichen. Bei diesem Programmausschnitt nehmen wir an, daß ein Bit der Information im Register AX wichtig sei und geben dabei im CL-Register an, welches dieses Bit ist. Enthält CL den Wert 8, so wird Bit 8 des Wertes in AX ausgewählt. Die Routine schiebt dabei den Maskenwert im BX-Register in die gewünschte Position, die durch den Wert im CL-Register bestimmt ist. Der darauffolgende AND-Befehl isoliert dann dieses ausgewählte Bit.

Damit das Beispiel allerdings funktioniert, müssen wir voraussetzen, daß der Wert im CL-Register im Bereich zwischen 0 und 15 liegt. Wir könnten dabei einen AND-

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.21 Shift Examples

 1                                    PAGE    ,132
 2                                    TITLE   Figure 4.21 Shift Examples
 3        0000                CODE    SEGMENT
 4                                    ASSUME  CS:CODE,DS:CODE
 5
 6                            ;-----------------------------------------
 7                            ; This routine takes the number in AX
 8                            ; multiplies it by 9, without using the
 9                            ; multiply instruction.
10                            ;-----------------------------------------
11
12        0000                MUL9    PROC    NEAR
13
14        0000  51                    PUSH    CX              ; Save the CX register on the stack
15        0001  50                    PUSH    AX              ; Temporary save of AX
16        0002  B1 03                 MOV     CL,3            ; Will shift by 3 positions
17        0004  D3 F8                 SAR     AX,CL           ; Multiply AX by 8
18        0006  8B C8                 MOV     CX,AX           ; AX*8 in CX
19        0008  58                    POP     AX              ; Original AX value
20        0009  03 C1                 ADD     AX,CX           ; AX now has 9 times original value
21        000B  59                    POP     CX              ; Recover the original CX value
22        000C  C3                    RET
23        000D                MUL9    ENDP
24
25                            ;-----------------------------------------
26                            ; This program fragment isolates the bit
27                            ; in the AX register specified by the number
28                            ; in CL.
29                            ;-----------------------------------------
30
31        000D  53                    PUSH    BX              ; Save BX on stack
32        000E  BB 0001               MOV     BX,1            ; Put a one in Bit 0 position
33        0011  D3 C3                 ROL     BX,CL           ; Move the mask bit
34        0013  23 C3                 AND     AX,BX           ; AND isolates the selected bit
35        0015  5B                    POP     BX              ; Recover BX value
36
37        0016                CODE    ENDS
38                                    END
```

Abbildung 4.21 Beispiele für Schiebebefehle

Befehl verwenden, um zu garantieren, daß sich nur die 4 niederwertigen Bits der Schiebezahl im CL-Register befinden. Der Befehl AND CL,0FH garantiert uns, daß der Zahlenwert im CL-Register zwischen 0 und 15 liegt. Wir könnten dieses Beispiel weiterhin modifizieren, um mehr als ein Bit aus dem gewünschten Wort zu isolieren. So könnten wir beispielsweise ein halbes Byte aus dem 16-Bit-Wort herauslösen, indem wir den ursprünglichen Maskenwert im BX-Register entsprechend verändern.

Stringbefehle

Einer der Bereiche im Befehlssatz des 8088, der besondere Aufmerksamkeit verdient, ist der Bereich der Stringbehandlung. Ein Zeichen- oder Zahlenstring ist dabei ganz allgemein ein Datentyp, den ein Programm als Einheit bearbeitet. Das Programm kann dabei einen String von einer Stelle an eine andere transportieren, kann ihn mit anderen Strings vergleichen oder kann in ihm nach bestimmten Werten suchen. Zeichenstrings sind dabei der gängigste Typ von Stringdaten. Ein Programm stellt ein Wort, einen Satz oder eine andere Datenstruktur durch eine Kette von Zeichen — einen Zeichenstring — im Speicher dar. Funktionen zum Editieren von Texten beispielsweise benützen dabei besonders oft Such- und Transportbefehle. Die Stringbefehle des 8088 erfüllen diese Aufgaben mit einem Minimum an Programmieraufwand und auch mit einem Minimum an Ausführungszeit.

Sehen wir uns als erstes das Konzept der Stringbearbeitung an. Ein Programm kann Stringoperationen sowohl auf Byte- als auch auf Wortbasis ausführen. Das heißt, die

einzelnen Elemente eines Strings können entweder 8 oder 16 Bits lang sein. Allerdings verwenden wir bei Stringbefehlen nicht die normalen Adressierungsarten, die wir bei den sonstigen Befehlen verwenden. Stringbefehle erfordern nämlich eine genau vorgeschriebene Art der Adressierung, die keine Variationen zuläßt. In Stringanweisungen werden die Operanden entweder über das Registerpaar DS:SI oder das Registerpaar ES:DI adressiert. Die Quelloperanden benutzen dabei das Registerpaar DS:SI, die Zieloperanden das Registerpaar ES:DI — daher auch die Namen Quellindex- und Zielindexregister. Alle Stringbefehle verfügen über eine Automatik, die nach Ausführung eines Befehls jeweils die Adressen erhöht. Ein String ist nämlich aus vielen Einzelteilen aufgebaut, die Stringbefehle ihrerseits können jedoch immer nur mit einem einzigen Element eines Strings arbeiten. Unser Programm muß sich also schrittweise, ein Element nach dem anderen, durch den String hindurcharbeiten. Das automatische Erhöhen oder Erniedrigen der Adressen ermöglicht dabei eine sehr schnelle Verarbeitung von Stringdaten. Das Richtungsflag im Statusregister kontrolliert außerdem die Richtung der Stringabarbeitung. Ist das Richtungsflag dabei auf 1 gesetzt, so nehmen die Adressen ab, ist das Richtungsflag auf 0 gesetzt, so werden die Adressen nach Ausführung eines Stringbefehls erhöht. Die Schrittweite beim Erhöhen oder Erniedrigen der Adressen wird durch die Art des Operanden bestimmt. Stringbefehle auf Byte-Basis verändern die Adressen nach dem jeweiligen Befehl um eins, während Stringbefehle auf Wortbasis die Adressen jeweils um zwei erhöhen. Auf diese Weise zeigen die verwendeten Registerpaare nach Ausführen einer Stringoperation immer auf das nächste Element im String.

Laden und Speichern

Die Programmliste in Abbildung 4.22 zeigt die verschiedenen Stringbefehle. Die einfachsten Befehle sind dabei „Laden String" (Load String — LODS) und „Speichern String" (Store String — STOS). Verwenden wir als Operanden für den Befehl LODS einen Byte-Operanden, so wird der Inhalt der über das Registerpaar DS:SI adressierten Speicherstelle in das Register AL geladen. Danach wird der Wert im Register SI um 1 verändert. Er wird in Abhängigkeit vom Richtungsflag entweder erhöht oder aber erniedrigt. Verwenden wir beim Befehl LODS einen Wortoperanden, so wird das AX-Register geladen und das SI-Register um 2 verändert. Der Befehl STOS arbeitet genau entgegengesetzt und speichert dann das im Register AL enthaltene Byte oder das im Register AX enthaltene Wort an die definierte Speicherstelle. Bei diesem Befehl bestimmt das Registerpaar ES:DI die Adresse der gewünschten Speicherstelle. Der Befehl verändert dabei das DI-Register entweder um 1 oder um 2 in Abhängigkeit vom Typ des jeweiligen Operanden.

In einem Assembler-Programm können wir den Befehl LODS und auch alle anderen Stringbefehle auf verschiedene Weisen angeben. Entweder wir spezifizieren den Operanden als Teil des Operationscodes oder wir lassen den Assembler die Art des zu bearbeitenden Stringelements selbst über die angegebenen Operanden herausfinden. In Abbildung 4.22 sehen wir, daß der Befehl

```
LODS    EXBYTE
```

den gleichen Ladebefehl für Byte-Strings erzeugt, wie es auch der Befehl

LODSB

tut.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 4.22  String Instructions

1                                     PAGE    ,132
2                                     TITLE   Figure 4.22  String Instructions
3         0000                        CODE    SEGMENT
4                                     ASSUME  CS:CODE,DS:CODE,ES:CODE
5         0000               EXBYTE   LABEL   BYTE
6         0000               EXWORD   LABEL   WORD
7         0000               EXBYTE1  LABEL   BYTE
8         0000               EXWORD1  LABEL   WORD
9
10        0000  AC                    LODS    EXBYTE            ; Load AL from DS:SI
11        0001  AD                    LODS    EXWORD            ; Load AX from DS:SI
12        0002  AC                    LODSB                     ; Load AL from DS:SI
13        0003  AA                    STOS    EXBYTE            ; Store to ES:DI from AL
14        0004  AB                    STOS    EXWORD            ; Store to ES:DI from AX
15        0005  AB                    STOSW                     ; Same as above
16        0006  F3/ AA                REP     STOSB             ; Store AL at ES:DI for CX times
17
18        0008  A4                    MOVS    EXBYTE1,EXBYTE    ; Move byte [ES:DI] <- [DS:SI]
19        0009  A5                    MOVS    EXWORD1,EXWORD    ; Move word [ES:DI] <- [DS:SI]
20        000A  A4                    MOVSB                     ; Move byte [ES:DI] <- [DS:SI]
21
22        000B  F3/ A5                REP     MOVSW             ; Move words CX times
23
24        000D  AE                    SCAS    EXBYTE1           ; Test AL w th [ES:DI]
25        000E  F3/ AE                REPE    SCASB             ; Test AL with [ES:DI] while =
26        0010  F2/ AF                REPNE   SCASW             ; Test AX with [ES:DI] while <>
27
28        0012  A7                    CMPS    EXWORD,EXWORD1    ; Compare word [DS:SI] to [ES:DI]
29        0013  F3/ A7                REPE    CMPSW             ; Compare words while equal and CX <> 0
30        0015  F2/ A6                REPNE   CMPSB             ; Compare bytes while not equal, CX <> 0
31
32        0017               CODE     ENDS
33                                    END
```

Abbildung 4.22 Stringbefehle

Im ersten Fall kann der Assembler selbst ermitteln, daß es sich um einen Byte-String handelt, da die Variable EXBYTE eine Bytevariable ist. Im zweiten Fall geben wir direkt an, daß es sich um einen Byte-Befehl handelt. In diesem Fall benötigt der Assembler dann keinen weiteren Operanden. Diese letztere Form wird sehr oft verwendet, da es hier nicht nötig ist, einen Variablennamen zu verwenden. Der String darf sich nämlich dann irgendwo im Speicher befinden, wobei die Adresse nicht festliegen muß und es demzufolge auch keinen Namen hierfür gibt. Beim Befehl STOS verhält es sich ähnlich. Um anzugeben, daß es sich um einen Wortstring anstelle eines Bytestrings handelt, schreiben wir einfach LODSW bzw. STOSW. In jedem Fall muß der Assembler wissen, ob es sich um einen Befehl für einen Byte- oder Wortstring handelt, da in den beiden Fällen jeweils verschiedener Maschinencode erzeugt wird. In Abhängigkeit von diesem Maschinencode wird später entschieden, um welchen Wert die Indexregister verändert werden.

Wollen wir die einfachen Befehle LODS bzw. STOS verwenden, so müssen wir immer einen Operanden angeben. Verfügen wir dagegen über kein passendes Label für unseren String, so können wir einfach die Befehle LODSB bzw. STOSB verwenden. Der Vorteil bei der Verwendung der Grundformen dieser Befehle liegt darin,

daß wir durch die Angabe eines Operandennamens den Assembler überprüfen lassen, ob der Typ des Operanden geeignet, und ob er richtig adressierbar ist. Da der Befehl LODS nur Daten im DS-Segment erreichen kann, muß vorher ein entsprechender ASSUME-Befehl für das passende Segment gegeben worden sein. Ebenso überprüft der Assembler bei Verwendung der Grundform des Befehls STOS die Adressierung über das ES-Segment. Sowohl die Grund- als auch die erweiterten Formen werden vom Assembler akzeptiert, doch sollten wir zumindest am Anfang häufigen Gebrauch von den Grundformen machen, um durch die Prüfroutinen des Assemblers mögliche Programmfehler zu verhindern.

Wiederholungsangabe

Es existiert noch eine weitere spezielle Anwendung der Stringbefehle. Für Stringbefehle gibt es nämlich ein spezielles Wiederholungspräfix. Dieses Präfix geht den Stringanweisungen voran und modifiziert ihren Ablauf, ähnlich den Segment-Präfixen bei Verwendung der direkten Segmentadressierung. Im Gegensatz dazu allerdings verwandelt das REP-Präfix den Stringbefehl in eine Schleife. Die Bezeichnung REP steht nämlich für „Repeat", also Wiederholung. Der 8088 verwendet dieses Präfix in Zusammenhang mit dem CX-Register, das die Anzahl der Wiederholungen für den jeweiligen Befehl enthält.

Verwenden wir als Beispiel den Befehl STOSB.

```
REP     STOSB
```

Dies ist eine spezielle Form des Befehls STOSB. Der Befehl wird solange ausgeführt, bis das Register CX durch schrittweises Erniedrigen schließlich bei Null angelangt ist. Dabei wird der Inhalt des Registers AL an die durch das Registerpaar ES:DI adressierte Speicherstelle gespeichert und dann das Register DI um jeweils 1 erhöht bzw. erniedrigt — gerade wie bei einem normalen STOSB-Befehl. Der Befehlszusatz REP erniedrigt außerdem das CX-Register und wiederholt, wenn sein Inhalt noch nicht 0 ist, den gesamten Befehl. Der Befehl wird im weiteren solange wiederholt, bis der Inhalt des CX-Registers Null erreicht.

Diese Fähigkeit verwandelt den normalen Speicherbefehl STOS zu einem Befehl, der dazu dienen kann, Speicherbereiche aufzufüllen. Dazu stellen wir den gewünschten Wert in das AL-Register, die Anzahl der aufzufüllenden Bytes ins CX-Register und die Adresse des gewünschten Blocks ins Registerpaar ES:DI und löschen schließlich noch das Richtungsflag. Der Befehl REP STOSB füllt dann den adressierten Speicherblock in der gewünschten Länge mit dem im Register AL enthaltenen Wert auf. Abbildung 4.23 zeigt ein solches Beispiel.

Allerdings gibt das Präfix REP keinen Sinn, wenn wir es im Zusammenhang mit dem Befehl LODS verwenden. Das Laden eines zusammenhängenden Zeichenstrings in den Akkumulator gestattet unserem Programm nämlich nicht, irgend etwas mit den solchermaßen geladenen Daten anzufangen. Jedoch können wir den Befehlszusatz REP mit den anderen Stringanweisungen sehr sinnvoll verknüpfen.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 4.23  Block Fill

1                                         PAGE    ,132
2                                         TITLE   Figure 4.23  Block Fill
3        0000                     CODE    SEGMENT
4                                         ASSUME  CS:CODE,DS:CODE,ES:CODE
5
6                                 ;--------------------------------------
7                                 ; This example fills the data area
8                                 ; BYTE_BLOCK with the value 01H
9                                 ;-----------------------------------
10
11       0000  BF 000B R                  MOV     DI,OFFSET BYTE_BLOCK    ; Address of data area
12       0003  B9 0032 90                 MOV     CX,BYTE_BLOCK_LENGTH    ; Number of bytes to fill
13       0007  B0 01                      MOV     AL,01H                  ; Fill character
14       0009  F3/ AA             REP     STOS    BYTE_BLOCK              ; Fill the block
15
16       000B      32 [           BYTE_BLOCK      DB      50 DUP(?)
17                     ??
18                         ]
19
20       = 0032                   BYTE_BLOCK_LENGTH       EQU     $-BYTE_BLOCK
21
22       003D                     CODE    ENDS
23                                        END
```

Abbildung 4.23 Auffüllen eines Speicherblocks

Stringtransport

Wir könnten nun versucht sein, die Befehle LODS und STOS dazu zu verwenden, Daten von einer Speicherstelle an eine andere zu transportieren; doch dazu gibt es einen besonderen Befehl, nämlich MOVS (Move String). Dieser Befehl arbeitet wie eine Kombination der beiden Befehle LODS und STOS. Er nimmt die durch das Registerpaar DS:SI adressierten Daten, stellt sie an die durch ES:DI adressierte Stelle und verändert zusätzlich sowohl das SI- als auch das DI-Register, damit diese dann auf den jeweils nächsten Punkt sowohl im Quell- als auch im Zielstring zeigen. Dies alles wird in einem einzigen Befehl bewirkt, wir benötigen dazu nicht einmal den Akkumulator als Zwischenspeicher. MOVS wirkt also wie eine Kombination aus LODS und STOS, nur daß er wesentlicher schneller und mit wesentlich weniger Seiteneffekten arbeitet.

Der Befehl MOVS arbeitet mit zwei Speicheroperanden. Zusammen mit einem weiteren Stringbefehl (CMPS) sind dies die beiden einzigen Anweisungen des 8088, die zwei Speicheroperanden verarbeiten können. Alle anderen Befehle verlangen nämlich, daß sich mindestens einer der beiden Operanden in einem Register befindet. Analog zu den Befehlen LODS und STOS arbeitet auch der Befehl MOVS sowohl auf Byte- wie auch auf Wortbasis. Da die Stringbefehle die Adressierungsart vorschreiben, dienen die angegebenen Operanden nur dazu, um den Typ des Strings festzustellen. Dazu müssen beide Operanden angegeben werden und beide müssen denselben Typ haben. Wir können den Typ der Stringoperation aber auch im Befehlscode spezifizieren, also MOVSB für Bytestrings und MOVSW für Wortstrings. Verwenden wir die Grundform MOVS, so überprüft der Assembler die korrekte Adressierung der Segmente und stellt auch den Typ der Operanden fest.

Kombinieren wir nun den Befehl MOVS mit dem Präfix REP, so erhalten wir einen mächtigen Blocktransportbefehl. Mit der gewünschten Anzahl im CX-Register und dem Richtungsflag zur Angabe der Transportrichtung bewegt der Befehl REP MOVS die Daten sehr schnell von einem Speicherbereich in einen anderen. Hat der Pro-

zessor einmal einen solchen Befehl begonnen, so werden die Daten mit der höchstmöglichen Geschwindigkeit transportiert. Während des Transports müssen nämlich keine anderen Befehle mehr ausgeführt werden, so daß der Prozessor voll für den Datentransport zur Verfügung steht.

Das Setzen des Richtungsflags ist dabei entscheidend für den korrekten Ablauf des Befehls REP MOVS. Wir haben die verschiedenen Aufgaben des Richtungsflags bereits in Kapitel 3 besprochen, besonders hinsichtlich der Stringtransportbefehle. Wir müssen uns genau an die dort festgelegten Richtlinien halten, besonders wenn sich Quell- und Zieloperand überlappen.

Suchen und Vergleichen

Die beiden restlichen Stringbefehle können wir in unserem Programm dazu verwenden, um Stringinformationen zu vergleichen. Der erste Befehl ist „Suchen im String" (Scan String — SCAS). Dieser Befehl vergleicht den im AL- oder AX-Register enthaltenen Wert mit der Speicherstelle, die durch das Registerpaar ES:DI adressiert ist. Der Befehl SCAS setzt dabei das Null-, Carry- und Überlaufsflag, um das Ergebnis des Vergleichs zwischen Akkumulator und Speicherstelle festzuhalten. Natürlich modifiziert der Befehl SCAS auch das DI-Register, damit dieses auf das nächste Element im String zeigt.

Im übrigen können wir beim Befehl SCAS das Präfix REP nicht verwenden, um beispielsweise einen längeren String zu durchsuchen. So wie der Befehl REP LODS keinen Sinn ergibt, erlaubt es auch der Befehl REP SCAS unserem Programm nicht, jeden Vergleich zu testen. Allerdings gibt es für diesen Spezialfall zwei Erweiterungen des Präfixes REP, nämlich REPE (Repeat while Equal) und REPNE (Repeat while Not Equal). Wie bei der Verwendung des normalen Präfixes REP laden wir als erstes das CX-Register mit der Länge unseres Strings. Haben wir nun das Präfix REPE angegeben, so wird der Suchbefehl solange ausgeführt, bis der Inhalt des AL- (oder AX-Registers) nicht mehr mit der adressierten Speicherstelle übereinstimmt oder der Inhalt des CX-Registers den Wert Null erreicht. Solange Akkumulator und Speicher übereinstimmen, wird der Befehl SCAS fortgeführt. Das Präfix REPNE arbeitet genau in der entgegengesetzten Weise. Die Suche wird dabei solange fortgeführt, bis der Inhalt des Akkumulators mit dem Inhalt des Speichers übereinstimmt.

Die Kombination des Befehls SCAS mit dem Präfix REPNE erlaubt es unserem Programm, eine sehr schnelle Tabellensuche durchzuführen. Um z.B. einen Eintrag in einer Tabelle zu finden, muß unser Programm jede Stelle in der Tabelle auf Gleichheit mit dem jeweiligen Argument überprüfen. In Abbildung 4.24 sehen wir, wie der Befehl SCAS diese Aufgabe erfüllt. Das Register AL enthält dabei das Suchargument. Die Tabelle SCAN_TABLE enthält die Werte, innerhalb deren wir suchen wollen, und das CX-Register die Länge der Tabelle. Der Befehl REPNE SCASB sucht nun solange in der Tabelle, bis ein Tabellenelement mit dem Akkumulatorinhalt übereinstimmt. In diesem Augenblick zeigt nun das DI-Register auf das nächstfolgende Byte in der Tabelle. Wir können nun den Offset des gefundenen Werts zum Tabellenanfang ganz einfach dadurch bestimmen, daß wir vom DI-Register 1 abziehen.

Diese Offsetinformation können wir zum Zugriff auf andere Tabellen verwenden, die beispielsweise irgendeine Antwort auf den gesuchten Wert enthalten könnten. Wichtig ist dabei der Befehl JE, der unserem Suchbefehl folgt. Es gibt zwei Möglichkeiten, zu diesem Befehl zu gelangen: entweder das Byte im String stimmt mit dem Inhalt des AL-Registers überein und die Bedingung für REPNE ist nicht länger erfüllt, oder aber der Wert im CX-Register hat Null erreicht, ohne daß eine Übereinstimmung in der Tabelle gefunden wurde. Den zweiten Fall können wir dabei durch geschicktes Programmieren verhindern. Was wir aber nicht verhindern können ist, daß für die Suche falsche Daten eingegeben wurden. Nach dem Suchbefehl springen wir in unserem Programm auf das Label FOUND, vorausgesetzt der Suchbefehl hat das Nullflag gesetzt. Dies garantiert uns, daß eine Übereinstimmung gefunden wurde. Hat das CX-Register dagegen den Wert Null erreicht, so war die letzte Suche nicht erfolgreich, folglich ist auch das Nullflag gelöscht.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.24  Table Scan


1                                                  PAGE    ,132
2                                                  TITLE   Figure 4.24  Table Scan
3        0000                              CODE    SEGMENT
4                                                  ASSUME  CS:CODE,DS:CODE,ES:CODE
5
6                                          ;--------------------------------------
7                                          ; This example scans a table of values until
8                                          ; the value in the AL register matches the table
9                                          ; entry.
10                                         ;--------------------------------------
11
12       0000  BF 000B R                           MOV     DI,OFFSET SCAN_TABLE     ; Address of table
13       0003  B9 000A 90                          MOV     CX,SCAN_TABLE_LENGTH     ; Length of scan table
14       0007  F2/ AE                      REPNE   SCASB                            ; Scan until match found
15       0009  74 00                               JE      FOUND                    ; If equal, then match
16                                         ; ...                                    ; Otherwise no match found
17       000B                              FOUND:
18                                         ;----- Program continues here . . .
19
20
21       000B  51 57 45 52 54 59           SCAN_TABLE      DB      'QWERTYUIOP'
22             55 49 4F 50
23       = 000A                            SCAN_TABLE_LENGTH       EQU     $-SCAN_TABLE
24
25       0015                              CODE    ENDS
26                                                 END
```

Abbildung 4.24 Tabellensuche

Der letzte Stringbefehl ist der Vergleichsbefehl (Compare Strings — CMPS). Er stellt wie der Suchbefehl eine Vergleichsoperation dar. Und so wie der Befehl MOVS arbeitet auch er mit zwei Speicheroperanden. Der Befehl CMPS vergleicht dabei den String an der Adresse DS:SI mit dem String an der Adresse ES:DI und setzt die Flags entsprechend. Wie beim Befehl SCAS können wir auch hier das Präfix REPE nicht verwenden, dagegen sind die Präfixe REPE und REPNE sehr empfehlenswert.

Abbildung 4.25 zeigt ein Beispiel für die Anwendung des Befehls CMPS. Dabei wird ein 5 Zeichen langer Eingabestring mit einer Tabelle von Zeichenstrings verglichen. Das Programm soll nun überprüfen, ob der Eingabestring mit einem Tabelleneintrag übereinstimmt. Das Ergebnis, im BX-Register enthalten, ist dann die Indexnummer des Strings, falls er gefunden wird. Wir verwenden dabei das Präfix REPE, so daß der Vergleichsbefehl solange ausgeführt wird, bis eines der Zeichen im Argument nicht mehr mit den Zeichen der Tabelle übereinstimmt. Passen alle fünf Zeichen aufeinander, so haben wir den gesuchten Eintrag gefunden. Der Befehl JE (Jump if Equal — Springe bei Gleichheit) überprüft das Ergebnis des Befehls CMPS. Wäre der Vergleichsbefehl wegen Ungleichheit beendet worden, hätte das Nullflag diese

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.25  String Compare

 1                                        PAGE    ,132
 2                                        TITLE   Figure 4.25  String Compare
 3       0000                      CODE   SEGMENT
 4                                        ASSUME  CS:CODE,DS:CODE,ES:CODE
 5
 6                                 ;-------------------------------------
 7                                 ; This example compares a five character
 8                                 ; string against a table of five character strings.
 9                                 ; The routine exits when it finds a match.
10                                 ;-------------------------------------
11
12       0000                      FIG4_25 PROC   NEAR
13       0000   BE 001B R                 MOV     SI,OFFSET ARGUMENT       ; Address of search parameter
14       0003   BF 0020 R                 MOV     DI,OFFSET COMPARE_TABLE  ; Table to be searched
15       0006   BB 0000                   MOV     BX,0                     ; BX will count table entries
16       0009                      COMPARE_LOOP:
17       0009   56                        PUSH    SI                       ; Save arg pointer
18       000A   57                        PUSH    DI                       ; Save table pointer
19       000B   B9 0005                   MOV     CX,5                     ; Compare 5 bytes
20       000E   F3/ A6             REPE   CMPS    ARGUMENT,COMPARE_TABLE   ; Compare the values
21       0010   5F                        POP     DI                       ; Recover pointers
22       0011   5E                        POP     SI
23       0012   74 06                     JE      FOUND                    ; Match has occurred
24       0014   83 C7 05                  ADD     DI,5                     ; Move to next table entry
25       0017   43                        INC     BX                       ; Adjust index to next
26       0018   EB EF                     JMP     COMPARE_LOOP             ; Go back and try again
27
28       001A                      FOUND:
29       001A   C3                        RET
30       001B                      FIG4_25 ENDP
31
32       001B   41 42 43 44 45     ARGUMENT       DB      'ABCDE'
33       0020                      COMPARE_TABLE  LABEL   BYTE
34       0020   51 57 45 52 54 50         DB      'QWERT','POIUY','ASDFG','LKJHG'
35              4F 49 55 59 41 53
36              44 46 47 4C 4B 4A
37              48 47
38       0034   5A 58 43 56 42 4D         DB      'ZXCVB','MNBVC','VWXYZ','ABCDE'
39              4E 42 56 43 56 57
40              58 59 5A 41 42 43
41              44 45
42
43       0048                      CODE   ENDS
44                                        END
```

Abbildung 4.25 Stringvergleich

Ungleich-Bedingung angezeigt. Wäre der Befehl CMPS jedoch beendet worden, weil das CX-Register Null erreicht, hätte das Nullflag Gleichheit angezeigt und das Programm wäre auf FOUND verzweigt. Seien Sie sich deshalb im Klaren darüber, daß an diesem Programm einige wesentliche Dinge fehlen, die es zu einem wirklich guten Programm machen. Beispielsweise ist der Fall nicht vorgesehen, daß eine Eingabe keinem der Tabelleneinträge entspricht. Jeder gute Programmierer wird Ihnen sagen, daß Sie die Behandlung solcher Ausnahmezustände immer vorsehen müssen.

Steuerbefehle

Steuerbefehle werden verwendet, um das gerade ablaufende Programm an einer anderen Stelle fortzuführen. Zu dieser Gruppe von Befehlen gehören Unterprogrammaufrufe und Sprungbefehle. Ein CALL-Befehl dient zum Aufrufen eines Unterprogramms, während Sprungbefehle die Steuerung an die gewünschte Stelle im Programm übertragen, ohne eine Rückkehradresse zu sichern. Die bedingten Sprungbefehle erlauben es dem Computer außerdem, zu „denken“. Bedingte Anweisungen können das Ergebnis vorausgehender Operationen testen und in Abhängigkeit davon den Fluß des Programms modifizieren. Gäbe es keine bedingten Sprungbefehle, wäre das Programmieren viel einfacher, aber auch wesentlich weniger produktiv.

Das erste, was wir bei den Steuerbefehlen besprechen müssen, ist die Adressierungsmethode zur Identifizierung des Sprungziels. Obwohl auch das Ziel einer Sprunganweisung eine Speicherstelle ist, also mit dem Ansprechen von Daten im Speicher verglichen werden kann, arbeiten diese Befehle doch gänzlich anders. Deshalb gibt es auch andere und bessere Wege, das jeweilige Sprungziel anzusprechen.

Near und Far

Ein Sprungbefehl verändert den Befehlszähler IP und möglicherweise auch das Code-Segmentregister CS. Dieses Registerpaar bezeichnet den nächsten auszuführenden Befehl. Mit anderen Worten, ein Sprungbefehl ist eine Sonderform des Datentransportes in ein Register bzw. Registerpaar. Und in der Tat wird von einigen Computern der Sprungbefehl in genau dieser Weise bearbeitet. Allerdings ist die Art, wie der 8088 das Registerpaar CS:IP lädt, völlig verschieden vom Vorgang des Ladens der übrigen Register.

Bevor wir fortfahren, müssen wir uns einige Definitionen in Erinnerung rufen. Verändert ein Sprungbefehl nur das IP-Register, ist es ein intrasegmentaler Sprung, weil er innerhalb der aktuellen Segmentgrenzen bleibt. Wir nennen ihn auch NEAR JUMP. Verändert der Sprungbefehl dagegen auch das CS-Register, bezeichnen wir ihn als intersegmental, oder FAR JUMP.

Die Vergabe der Attribute NEAR und FAR für Steuerbefehle ist Aufgabe des Assemblers. Jedes Label in einem Programm hat eines der Attribute NEAR oder FAR, so wie Datenwerte die Attribute BYTE oder WORD besitzen. Etliche der Beispiele in diesem Kapitel enthalten Prozeduren, die als Attribut in der PROC-Anweisung NEAR enthalten. Dies bedeutet, daß das mit PROC bestimmte Label (der Name der Prozedur) das Attribut NEAR hat. Der Assembler verwendet diese Information, um zu bestimmen, welche Art von Sprungbefehl er generieren soll, um an eine solche Stelle zu verzweigen bzw. zu gelangen. Da die meisten Prozeduren außerdem Unterprogramme sind, legen die Attribute NEAR und FAR der PROC-Anweisung zudem fest, welche Art von RET-Befehl für das Unterprogramm erzeugt werden muß. Ein CALL auf eine FAR-Prozedur sichert sowohl CS wie auch IP, während ein CALL auf ein NEAR-Unterprogramm nur den IP-Wert in den Stack sichert. Der RET-Befehl muß dann seinerseits eine Angabe darüber enthalten, auf welche Weise die jeweilige Routine die Steuerung empfing, um wieder an die richtige Stelle zurückkehren zu können.

Sprungadressen

Enthält der Befehl die Zieladresse für den Sprung oder Unterprogrammaufruf im Befehlscode selbst (ähnlich den Daten in Befehlen mit Direktwerten), bezeichnen wir ihn als direkten Sprung. Befindet sich die Adresse des Sprungziels jedoch in einem Register oder in einer Speicherstelle, sprechen wir von einem indirekten Sprung, denn der Befehl muß die Zieladresse erst von einer anderen Stelle abholen.

Das Programm kann also nicht direkt, sondern erst über einen Zwischenschritt an die gewünschte Stelle verzweigen.

Es gibt zwei Wege, die Zieladresse eines Sprungs zu berechnen. Beinhaltet der Befehl selbst den Adresswert, nennen wir ihn absoluten Sprung. Es ist also ein Sprung an eine absolute Adresse. Der Befehl könnte das Sprungziel aber auch als Distanz zum aktuellen Wert das Befehlszählers angeben. Diese Methode, die dem Offset beim Indizieren von Datenwerten entspricht, bezeichnen wir als relativen Sprung.

Ein zusätzlicher Pluspunkt für relative Sprünge ist, daß ein Programm oft an Stellen springt, die nicht sehr weit vom Ursprungspunkt entfernt sind. Der Sprungbefehl kann dann ein 1-Byte Displacement benutzen. Behandeln wir dieses Displacement als Zahl im Zweierkomplement, dann kann der 2-Byte lange relative Sprungbefehl (ein Byte für den Befehlscode und ein Byte für das Displacement) innerhalb des Programms 127 Bytes vorwärts bzw. 128 Bytes rückwärts springen. Der 8088 verfügt über zwei Arten von relativen Sprungbefehlen: einer hat ein 1-Byte Displacement, der andere ein 2-Byte Displacement.

Alle bedingten Sprungbefehle des 8088 verwenden ein 1-Byte Displacement. Dies kann manchmal ungünstig sein, etwa wenn wir an eine 150 Bytes entfernte Stelle springen wollen. In solchen Fällen müssen wir jeweils ein Paar von Sprungbefehlen verwenden, einen bedingten und einen unbedingten. Ein Beispiel für diese Methode werden wir noch später zeigen. Im Normalfall minimiert jedoch das 1-Byte Displacement der bedingten Sprünge des 8088 den Speicherplatzbedarf für ein Programm.

Zur Bestimmung des Ziels eines relativen Sprungs wird vom 8088 einfach das Displacement auf den Wert im IP-Register addiert, den dieses nach Ausführen des Befehls enthält. Abbildung 4.26 zeigt mehrere Beispiele für relative Sprungbefehle. Ist die direkt auf den Befehl folgende Stelle das Sprungziel, so ist das Displacement 0. Um wieder auf sich selbst zu springen, wäre das Displacement −2. Mit einem 2-Byte Displacement können wir irgendwohin im Abstand von −32,768 bis 32,767 Bytes zum aktuellen Stand des IP-Registers springen.

Unbedingte Steuerbefehle

Ein unbedingter Befehl ist ein Befehl, der die Steuerung an einen angegebenen Punkt jedesmal überträgt, wenn er ausgeführt wird. Im Gegensatz dazu überprüft ein bedingter Befehl jeweils den Zustand des Rechners, um festzustellen, ob er die Steuerung übertragen soll oder nicht. Es gibt zwei Arten von unbedingten Steuerbefehlen: Sprünge und Unterprogrammaufrufe.

Alle CALL-Befehle sind unbedingt. In Abbildung 4.27 sehen wir die verschiedenen CALL-Befehle. Der intrasegmentale oder NEAR CALL gibt dabei einen neuen Wert für das IP-Register an und speichert den alten Wert von IP als Rückkehr- oder Returnadresse in den Stack. Der intersegmentale oder FAR CALL enthält neue Werte für CS- und IP-Register und sichert auch beide in den Stack. Ein direkter NEAR

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.26   Relative Jumps

1                                         PAGE    ,132
2                                         TITLE   Figure 4.26  Relative Jumps
3
4       0000                      CODE    SEGMENT
5                                         ASSUME  CS:CODE
6
7                                 ;----- This example shows the code offsets for relative jumps
8
9       0000                      START:
10      0000  EB FE                       JMP     START             ; Jump to itself
11      0002  EB 00                       JMP     SHORT LABEL_ONE ; Jump to next location
12      0004                      LABEL_ONE:
13      0004  EB 01 90                    JMP     LABEL_TWO         ; Forward jump w/o SHORT
14      0007                      LABEL_TWO:
15      0007  E9 0200 R                   JMP     LABEL_THREE       ; Long jump to far away
16
17                                ;. . . Use ORG to simulate far away
18
19      0200                              ORG     200H
20      0200                      LABEL_THREE:
21
22      0200                      CODE    ENDS
23                                        END
```

Abbildung 4.26 Relative Sprünge

CALL entspricht einem relativen Sprung mit einem 2-Byte Displacement. Alle anderen CALL-Befehle sind absolute Sprünge. Der direkte FAR CALL benötigt ein 4-Byte Operandenfeld, um die neuen Werte sowohl für das CS- als auch das IP-Register zu spezifizieren. Die indirekten Sprünge benützen das mod-r/m-Adressbyte, um einen Register- bzw. Speicheroperanden festzulegen. Dieser Operand enthält dann die eigentliche Unterprogrammadresse. Indirekte NEAR CALL-Befehle laden einen Wortoperanden in das IP-Register, FAR CALL-Befehle laden einen Doppelwortoperanden in das Registerpaar CS:IP. Dabei wird das erste Wort in IP und das zweite in CS geladen. Wird ein Register als Operand für einen indirekten und intersegmentalen CALL angegeben, so sind die Ergebnisse der Befehlsausführung nicht vorhersehbar. Der 8088 nimmt nämlich dann den neuen Wert für das CS-Register von irgendwoher. Das beste ist deshalb, diese Variante des Befehls möglichst nicht zu versuchen.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.27   Call Instructions

1                                         PAGE    ,132
2                                         TITLE   Figure 4.27  Call Instructions
3
4       0000                      FAR_SEG SEGMENT
5                                         ASSUME  CS:FAR_SEG
6       0000                      FAR_LABEL       PROC    FAR
7       0000  CB                          RET
8       0001                      FAR_LABEL       ENDP
9       0001                      FAR_SEG ENDS
10
11      0000                      CODE    SEGMENT
12                                        ASSUME  CS:CODE
13
14      0000  0000 ---- R         INDIRECT_FAR    DD      FAR_LABEL
15      0004  0018 R              INDIRECT_NEAR   DW      NEAR_LABEL
16
17      0006  E8 0018 R                   CALL    NEAR_LABEL              ; Relative call
18      0009  9A 0000 ---- R              CALL    FAR_LABEL               ; Absolute call
19
20      000E  2E: FF 16 0004 R            CALL    INDIRECT_NEAR           ; Indirect near call
21      0013  2E: FF 1E 0000 R            CALL    INDIRECT_FAR            ; Indirect far call
22
23      0018                      NEAR_LABEL      PROC    NEAR
24      0018  C3                          RET
25      0019                      NEAR_LABEL      ENDP
26      0019                      CODE    ENDS
27
28                                        END
```

Abbildung 4.27 CALL-Befehle

Zu den CALL-Befehlen gibt es entsprechende Rückkehr- bzw. Returnbefehle (RET). Alle Returns sind indirekte Sprünge, da sie die jeweilige Adresse aus der Spitze des Stacks entnehmen. Ein intrasegmentaler Return nimmt ein einzelnes Wort aus dem Stack und überträgt es in das IP-Register, während ein intersegmentaler Return zwei Wörter aus dem Stack entnimmt, wobei das Wort von der niedrigeren Adresse in das IP- und das Wort von der höheren Adresse in das CS-Register übertragen wird.

Sowohl NEAR- als auch FAR-Returnbefehle können wir im Programm durch die Angabe eines Bytezählers modifizieren. Nach Entnahme der Rückkehradresse(n) aus dem Stack addiert der jeweilige Returnbefehl diesen Wert auf den Stackpointer. Der Befehl erlaubt es also einem Unterprogramm, ohne eigene Stack-Lesebefehle Parameter aus dem Stack zu entfernen. Dies verstärkt die Bedeutung des Stack als Mittel zur Parameterübergabe an Unterprogramme. Wir haben dieses Konzept bereits an früherer Stelle in diesem Kapitel im Abschnitt „Stackbefehle" besprochen.

Die unbedingten Sprungbefehle (JMP) entsprechen in ihren Adressiermöglichkeiten den CALL-Befehlen. Doch gibt es zusätzlich einen Sprungbefehl, der ein 1-Byte Displacement für einen relativen NEAR-Sprung benützt. In diesem Fall gibt es keine entsprechende CALL-Anweisung, da sich Unterprogramme nur sehr selten in unmittelbarer Nähe der aufrufenden Stelle befinden. Für alle Sprungbefehle werden die den CALL-Befehlen entsprechenden Methoden zur Adressenerzeugung angewandt.

Eine Bemerkung noch über Codeoptimierung und die Arbeitsweise des Assemblers. Wenn der Assembler seinen ersten Durchlauf durch das Quellprogramm macht und den einzelnen Befehlen Adressen zuteilt, muß er entscheiden, ob er die 2-Byte- oder 3-Byte-Version des JMP-Befehls generiert. Wird dabei ein Sprung nach rückwärts ausgeführt, d.h. an eine Stelle, deren Adresse der Assembler bereits kennt, kann er leicht das korrekte Displacement bestimmen. Der Assembler weiß nämlich zu diesem Zeitpunkt bereits, ob sich der auszuführende Sprung innerhalb der Reichweite des kurzen Displacements befindet. Wird dagegen ein Sprung nach vorwärts ausgeführt, d.h. zu einem Label, das der Assembler noch nicht kennt, so muß er davon ausgehen, daß die zu überspringende Entfernung größer als 128 Bytes sein wird. Es wird in diesem Fall also die lange Form des Sprungbefehls erzeugt. Der Assembler muß nämlich von den ungünstigsten Bedingungen ausgehen, da er später nicht noch einmal an diese Stelle zurückkehren und den Befehl vergrößern kann. Er ersetzt jedoch den 3-Byte-Befehl durch einen 2-Byte-Befehl und eine 1-Byte Nulloperation, wenn er später herausfindet, daß das Sprungziel sich innerhalb der Reichweite des kurzen Displacements befindet. Da der kurze Sprung etwas schneller ausgeführt wird, bedeutet dies eine geringfügige Zeitersparnis, der Maschinencode bleibt aber immer noch länger als notwendig.

Weiß der Programmierer jedoch, daß sich das Sprungziel innerhalb der Reichweite eines kurzen Sprungbefehls befinden wird, so kann er dies dem Assembler mit folgender Anweisung mitteilen:

```
JMP     SHORT LABEL
```

Das Attribut SHORT teilt dem Assembler mit, daß er die kurze (SHORT) Form des Sprungbefehls erzeugen soll, obwohl er zu diesem Zeitpunkt das Sprungziel noch nicht kennt. Macht der Programmierer hier einen Fehler, und der Sprung sollte tatsächlich ein langer Sprungbefehl sein, so gibt der Assembler eine Fehlermeldung aus. Abbildung 4.26 zeigt ein Beispiel für die Angabe SHORT.

In Abbildung 4.28 sehen wir, wie ein Programm unter Verwendung indirekter Sprungbefehle eine Sprungtabelle benützen kann. Das Beispiel wählt in Abhängigkeit vom Argumentwert im AL-Register unter einer Anzahl von Routinen aus. Auf ähnliche Weise könnten wir auch Unterprogramme aufrufen. Dies wäre die Assemblerentsprechung zur CASE-Anweisung, wie wir sie aus einigen höheren Programmiersprachen kennen.

Bedingte Sprünge

Wir können die bedingten Sprünge in zwei Gruppen aufteilen: das Testen von Flags, wobei die Ergebnisse einer vorausgehenden arithmetischen oder logischen Operation ausgewertet werden und das Testen von Schleifen, wobei die Anzahl der Durchläufe durch einen bestimmten Programmabschnitt überprüft wird. Alle bedingten Sprünge haben ein 1-Byte Displacement. Soll ein bedingter Sprung ein Ziel ansprechen, das mehr als 128 Bytes entfernt ist, müssen wir eine besondere Konstruktion verwenden. Als Beispiel wollen wir einmal annehmen, daß unser Programm beim Label ZERO weiterfahren soll, wenn das Nullflag gesetzt ist. Dieses Label ist mehr als 128 Bytes entfernt. Um das Problem zu lösen, würde die Befehlsfolge wie folgt lauten:

```
          JNZ    CONTINUE
          JMP    ZERO
CONTINUE:
```

```
The IBM Personal Computer Assembler 01-01-83         PAGE    1-1
Figure 4.28  Branch Table

 1                                          PAGE    ,132
 2                                          TITLE   Figure 4.28  Branch Table
 3
 4      0000                        CODE    SEGMENT
 5                                          ASSUME  CS:CODE
 6
 7                                  ;----------------------------------------
 8                                  ; This example branches to a routine
 9                                  ;  based on the value in the AL register.
10                                  ;  The AL value is the index into the
11                                  ;  routines that can execute.
12                                  ;----------------------------------------
13
14      0000  2A FF                         SUB     BH,BH                 ; Zero to BH register
15      0002  8A D8                         MOV     BL,AL                 ; Index value into BX
16      0004  D1 E3                         SHL     BX,1                  ; * 2 for word index
17      0006  2E: FF A7 000B R              JMP     CS:[BX + BRANCH_TABLE] ; Indirect branch
18
19      000B                        BRANCH_TABLE    LABEL   WORD
20      000B  0011 R                        DW      ROUTINE_ONE
21      000D  0011 R                        DW      ROUTINE_TWO
22      000F  0011 R                        DW      ROUTINE_THREE
23                                  ; . . .
24
25      0011                        ROUTINE_ONE     LABEL   NEAR
26      0011                        ROUTINE_TWO     LABEL   NEAR
27      0011                        ROUTINE_THREE   LABEL   NEAR
28
29      0011                        CODE    ENDS
30                                          END
```

Abbildung 4.28 Sprungtabelle

Das Programm benützt also einen bedingten Sprung für die entgegengesetzte Bedingung. Ein unbedingter Sprung, der ein Displacement bis zu 32,768 enthalten kann, führt dann zum Label ZERO, während die Bedingung „nicht Null" zum Label CONTINUE führt. Ist allerdings das Minimieren der Programmgröße eines Ihrer Ziele, sollten Sie diese Methode vermeiden, denn hier wird ein bedingter Sprungbefehl zu einer 5-Byte Befehlsfolge. Manchmal genügt es schon, das Programm ein wenig umzustellen, um das gewünschte Label in die Reichweite eines Sprungbefehls zu bekommen. Allerdings macht es in den meisten Fällen überhaupt nichts aus, wie lang das Programm ist, solange es nur in den Speicher paßt und funktioniert. Sollten Sie jedoch beispielsweise versuchen, ein Programm für einen ROM von fester Größe zu schreiben, dann könnten sie von solchen Überlegungen betroffen sein. Normalerweise ist aber der Nutzen einer solchen Programmänderung den Aufwand nicht wert.

Testen der Bedingungscodes

Die erste Gruppe der bedingten Sprungbefehle testet den aktuellen Zustand des Flagregisters. Der Befehl verzweigt dann in Abhängigkeit von den jeweiligen Bedingungscodes. Die bedingten Sprungbefehle verändern jedoch in keinem Fall die Flags, sie prüfen nur ihren Zustand. Normalerweise hat ein vorausgehender arithmetischer oder logischer Befehl die Flags gesetzt. Im folgenden Beispiel gehen wir davon aus, daß ein Vergleichsbefehl (CMP) die Flags gesetzt hat.

Abbildung 4.29 zeigt die bedingten Sprungbefehle und die Flags, die von ihnen überprüft werden. In der Abbildung sehen wir zeilenweise die bedingten Sprungbefehle und die fünf Statusflags als Spalten. Ein X in der jeweiligen Position gibt an, daß der Befehl dieses Flag nicht prüft. „0" bedeutet, daß das Flag nicht gesetzt sein darf, damit die Bedingung erfüllt ist und der Sprung ausgeführt wird. Und „1" bedeutet schließlich, daß das Flag gesetzt sein muß, damit der Sprung ausgeführt wird. Bei einigen Tabelleneinträgen sehen wir sogar eine Formel, die erfüllt sein muß, damit der Sprung ausgeführt werden kann. Dies sind die vier arithmetischen Sprünge, die wir später genauer besprechen werden.

Abbildung 4.29 teilt weiterhin die bedingten Sprungbefehle in drei Abschnitte: Befehle, die direkt ein Flag testen; Befehle, die einen vorzeichenlosen arithmetischen Vergleich durchführen; und schließlich Befehle, die einen arithmetischen Vergleich mit Vorzeichen durchführen.

Abbildung 4.29(a) zeigt die einzelnen Flag-Tests. Ein bedingter Sprung kann jedes der fünf Flags direkt auf 0 oder 1 testen. In der Abbildung sehen wir den Test des Carryflags bei den vorzeichenlosen arithmetischen Vergleichen, da dieses Flag auch arithmetische Bedeutung besitzt. Halten wir bei dieser Gelegenheit fest, daß viele Sprungbefehle mit mehr als einem Assemblerbefehl dargestellt werden können, selbst wenn sie den gleichen Test durchführen. So lautet der Test des Nullflags beispielsweise JZ (Jump if Zero). Der Befehl JE (Jump if Equal) erzeugt denselben Maschinencode. Die folgende Befehlsfolge zeigt den Grund:

```
CMP     AX,BX
JE      LABEL
```

Der Befehl CMP subtrahiert BX von AX und setzt die Flags entsprechend dem Ergebnis. Da das Ergebnis Null ist, wenn die beiden Operanden gleich sind, zeigt das Nullflag also Gleichheit an. Entsprechend ist JNZ (Jump if Not Zero) identisch zu JNE (Jump if Not Equal). JP (Jump if Parity) entspricht JPE (Jump if Parity Even) und JNP (Jump if No Parity) entspricht dann JPO (Jump if Parity Odd).

Bedingte Sprünge	OF	CY	Flags Z	P	S	Kommentar
JE/JZ	X	X	1	X	X	
JP/JPE	X	X	X	1	X	
JO	1	X	X	X	X	
JS	X	X	X	X	1	
JNE/JNZ	X	X	0	X	X	
JNP/JPO	X	X	X	0	X	
JNO	0	X	X	X	X	
JNS	X	X	X	X	0	
			(a)			
JL/JNGE	a	X	X	X	b	a NEQ b
JLE/JNG	a	X	1	X	b	Z OR (A NEQ B)
JNL/JGE	a	X	X	X	b	a = b
JNLE/JG	a	X	0	X	b	(NOT Z) AND (a=b)
			(b)			
JB/JNAE/JC	X	1	X	X	X	
JBE/JNA	X	1	1	X	X	CY OR Z
JNB/JAE/JN	X	0	X	X	X	
JNBE/JA	X	0	0	X	X	(NOT CY) AND (NOT Z)
			(c)			

Abbildung 4.29 Flagtest bei bedingten Sprüngen. (a) Flagtest; (b) Arithmetik mit Vorzeichen; (c) vorzeichenlose Arithmetik

Arithmetische Vergleiche mit Vorzeichen sind die Grundlage für die nächste Gruppe von bedingten Befehlen, wie wir sie in Abbildung 4.29(b) sehen. Wir können dabei vier Bedingungen testen: „kleiner", „kleiner gleich", „größer", und „größer gleich". Die anderen vier Anweisungen sind nur die Negation der bereits erwähnten. Der Assembler verwendet „größer" und „kleiner" für vorzeichenbehaftete Arithmetik. Im nächsten Abschnitt werden wir sehen, daß der Assembler für vorzeichenlose arithmetische Vergleiche „über" und „unter" verwendet.

Sie werden arithmetische Vergleiche besser verstehen, wenn wir sie zusammen mit dem CMP-Befehl anwenden:

```
CMP    AX,BX
JL     LABEL
```

Der Sprung wird ausgeführt, wenn AX kleiner als BX ist. Wir können die Kombination von Vergleich und Bedingungstest als einen Befehl lesen. Zuerst kommt der Ziel-

operand, dann der Vergleichsoperator, und schließlich der Quelloperand. Hier ein weiteres Beispiel:

```
CMP    CX,WORD_IN_MEMORY
JNLE   LABEL
```

Wir lesen dies als „Springe, falls CX nicht kleiner oder gleich dem Inhalt der Speicherstelle WORD_IN_MEMORY ist". Wir können diese Technik verwenden, um den Sinn jedes arithmetischen Sprungs zu bestimmen, sei er nun mit oder ohne Vorzeichen.

Wie Abbildung 4.29(b) zeigt, testen die arithmetischen Vergleiche nicht direkt ein einzelnes Flag. Und in der Tat überprüft jeder Befehl eine Kombination aus Überlauf-, Vorzeichen- und möglicherweise auch Nullflag. So verlangt beispielsweise JL (Jump if Less), daß Vorzeichen- und Überlaufflag verschiedene Werte haben. Haben sie den gleichen Wert, so war der erste Operand nicht kleiner als der zweite. Sehen wir uns deshalb diese Operation noch ein wenig genauer an, um den Ablauf von arithmetischen Vergleichen mit Vorzeichen zu verstehen.

Wenn zwei Zahlen mit Vorzeichen verglichen werden, gibt es vier Kombinationen aus Vorzeichen- und Überlaufflag. Wir werden uns jede dieser Kombinationen ansehen, um festzustellen, was geschah und welche Beziehung die Operanden zueinander haben, um gerade dieses Ergebnis zu erzeugen. In allen Fällen gehen wir dabei davon aus, daß ein vorausgehender CMP-Befehl die Flags gesetzt hat. Erinnern wir uns, daß dieser Befehl die beiden Operanden subtrahiert und entsprechend dem Ergebnis dann die Flags setzt.

Vorzeichen (Sign — S) = 0, Überlauf (Overflow — O) = 0

S = 0 bedeutet, daß das Ergebnis der Subtraktion positiv war. O = 0 gibt an, daß kein Überlauf auftrat, das Resultat im Zweierkomplement also korrekt ist. Die Subtraktion zweier Zahlen mit einem positiven Ergebnis bedeutet, daß die erste Zahl größer war als die zweite, also besteht die Beziehung „größer". Allerdings ergibt auch die Subtraktion zweier gleich großer Zahlen ein positives Ergebnis, so daß die Bedingung S = 0, O = 0 tatsächlich für die Beziehung „größer gleich" steht.

S = 1, O = 0

In diesem Falle steht O = 0 für ein korrektes Ergebnis und S=1 für negativ. Um zu einem negativen Ergebnis zu gelangen, muß die größere Zahl von der kleineren abgezogen worden sein. Die Beziehung lautet also „kleiner".

S = 0, O = 1

Hier zeigt O = 1 an, daß das Ergebnis nicht korrekt ist — d.h. es ist nicht mehr darstellbar. Dies bedeutet, daß die Addition zweier positiver Zahlen ein negatives Ergebnis produziert hat, oder auch umgekehrt. Im Falle unseres Vergleiches zeigt dies an, daß das Vorzeichen des Ergebnisses nicht mehr korrekt ist. Das Ergebnis des Vergleichs lautet also eigentlich S = 1, O = 0, also „kleiner".

S = 1, O = 1

Wiederum zeigt O = 1 ein unkorrektes Ergebnis an. Die Subtraktion muß also eine sehr große positive Zahl ergeben haben, die Beziehung lautet deshalb „größer".

Auch das Nullflag spielt bei einigen Vergleichen eine Rolle. Beispielsweise ist die Bedingung für JLE (Jump if Less than or Equal) dann erfüllt, wenn entweder die Bedingung für „kleiner" (Vorzeichen- und Überlaufflag sind nicht gleich) oder die Bedingung für „Null" (Nullflag gesetzt) erfüllt ist. Diese drei Flags erlauben dem 8088 also den Test aller möglichen Kombinationen von Zahlen mit Vorzeichen.

Der letzte Teil der Tabelle, Abbildung 4.29(c) zeigt die Tests für vorzeichenlose Arithmetik. So wie mit dem Vorzeichen gibt es vier mögliche Kombinationen der beiden Operanden, die der Prozessor testen kann. Der Assembler verwendet dabei die Bezeichnungen „über" und „unter" für den Vergleich, um zugleich anzuzeigen, daß es sich um vorzeichenlose Arithmetik handelt. Die Designer des Befehlssatzes hatten sicherlich bei der Konzipierung dieser Möglichkeiten die Adressberechnung vor Augen, denn es gibt keine negativen Adressen. „Über" und „unter" bedeuten also in etwa Positionen innerhalb des Speichers, während „größer" und „kleiner" sich auf Zahlen (mit Vorzeichen) beziehen. Wichtig ist dabei nur, daß der Befehl, der im Assemblerprogramm niedergelegt wird, auch der ist, der tatsächlich ausgeführt wird — ohne Rücksicht auf den Typ der zu vergleichenden Operanden. Beispielsweise kann ein Programm zwei Zahlen mit Vorzeichen vergleichen und dann den Befehl JA (Jump if Above) anwenden. Der Prozessor führt den bedingten Sprung in Abhängigkeit von der Beziehung der beiden Zahlen zueinander aus und betrachtet sie dazu als vorzeichenlos. Es ist also Sache des Programmierers, den richtigen bedingten Befehl zu verwenden.

Beim Vergleich von vorzeichenlosen Zahlen berücksichtigt der 8088 nur zwei Flags. Das Carryflag zeigt an, welche der Zahlen die größere ist. Ein Vergleich setzt das Carryflag, wenn der erste Operand unter dem zweiten liegt. Er löscht das Carryflag, wenn der erste Operand über oder gleich dem zweiten ist. Das Nullflag gibt in diesem Fall an, welche der beiden Möglichkeiten die richtige ist.

Wir lesen vorzeichenlose Vergleiche genauso wie solche mit Vorzeichen, z.B.:

```
CMP   AX,BX
JA    LABEL
```

Das Programm springt auf LABEL, wenn AX „über" BX liegt. Der bedingte Sprung wird immer ausgeführt, wenn nach einem Vergleich die beschriebene Beziehung zwischen dem ersten und dem zweiten Operanden besteht.

Schleifensteuerung

Etliche bedingte Sprunganweisung sind zur Steuerung von Programmschleifen gedacht. Da Schleifen in Programmen sehr häufig verwendet werden, ist eine wirksame Kontrolle solcher Schleifen sehr wünschenswert. Abbildung 4.30 zeigt vier Befehle, die die Schleifensteuerung im 8088-Assembler erheblich vereinfachen.

So wie Stringbefehle das CX-Register als Zähler verwenden, verwenden die LOOP-Befehle das CX-Register als Schleifenzähler. Alle diese Befehle verwenden das CX-Register implizit als Zähler für die Anzahl der Schleifendurchläufe. Der LOOP-Befehl ist dabei der einfachste. Er erniedrigt jeweils das CX-Register und überträgt die Steuerung wieder an die angegebene Stelle, solange der Inhalt von CX noch nicht 0 ist. Ergibt die Subtraktion vom CX-Register jedoch Null, wird der Sprung nicht ausgeführt, sondern die Verarbeitung mit dem nächstfolgenden Befehl fortgeführt.

Die aufgeführte Befehlsfolge zeigt die normale Verwendung der LOOP-Anweisung.

```
        MOV     CX,LOOP_COUNT
BEGIN_LOOP:
        ; . . . auszuführende Programmschleife
        LOOP            BEGIN_LOOP
```

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 4.30  Loop Instructions

1                                               PAGE    ,132
2                                               TITLE   Figure 4.30  Loop Instructions
3
4       0000                            CODE    SEGMENT
5                                               ASSUME  CS:CODE
6
7                                       ;--------------------------------
8                                       ; This example shows the loop instructions.
9                                       ;  The instructions here do not form an
10                                      ;  executable program.
11                                      ;--------------------------------
12
13      0000  E3 06                             JCXZ    END_OF_LOOP     ; No loop if CX already zero
14
15      0002                            BEGIN_LOOP:
16
17                                      ; . . . Body of Loop
18
19      0002  E2 FE                             LOOP    BEGIN_LOOP      ; Jump until CX is zero
20
21                                      ; . . . Or, if testing a condition
22
23      0004  E, FC                             LOOPE   BEGIN_LOOP      ; Jump if equal and CX not zero
24
25                                      ; . . . Or
26
27      0006  E0 FA                             LOOPNE  BEGIN_LOOP      ; Jump if not equal and CX not zero
28
29      0008                            END_OF_LOOP:
30
31      0008                            CODE    ENDS
32                                              END
```

Abbildung 4.30 Schleifenbefehle

Dabei legen wir die Anzahl der Schleifendurchläufe vor Ausführung der Programmschleife im CX-Register ab. Hierauf wird die eigentliche Schleife ausgeführt, gefolgt von einem LOOP-Befehl. Diese Anweisung erniedrigt den Schleifenzähler um eins, entsprechend der gerade erfolgten einmaligen Ausführung der Schleife. Enthält CX nun Null, fährt das Programm mit der nächstfolgenden Anweisung fort. Ist der Zähler dagegen noch nicht Null, geht die Steuerung zurück an den Anfang der Schleife, um einen weiteren Durchlauf zu beginnen. Die Anzahl der ausgeführten Schleifendurchläufe entspricht genau der Zahl, die ursprünglich ins CX-Register geladen wurde. Wichtig ist dabei jedoch, festzuhalten, daß jede Veränderung von CX während des Schleifendurchlaufs dazu führt, daß die Anzahl der auszuführenden Durchläufe nicht mehr mit dem ursprünglich in CX enthaltenen Wert übereinstimmt.

Die gezeigte Methode funktioniert sowohl wenn die Anzahl der Schleifendurchläufe bereits zur Assemblierungszeit bekannt ist (wie im gezeigten Beispiel, wo LOOP_ COUNT ein Direktwert ist), als auch wenn die Anzahl der Durchläufe erst zur Ausführungszeit des Programms festgelegt wird. Entsteht allerdings bei der Berechnung der Durchläufe der Wert 0, so wird die Schleife 65,536 mal ausgeführt. Führt der 8088 nämlich in diesem Fall den ersten LOOP-Befehl aus, wird von CX eins abgezogen, wodurch der Wert 0FFFFH entsteht. Da CX nun ungleich Null ist, wird die Schleife wiederholt. Die Behandlung des Wertes Null im Schleifenzähler stellt also einen Sonderfall dar.

Dieser Sonderfall wird vom Befehl JCXZ (Jump if CX is Zero) behandelt. Dieser Befehl testet den aktuellen Inhalt des CX-Registers und verzweigt, wenn dieser Wert Null ist. Dabei wird keines der Bedingungsflags geprüft oder verändert. Das folgende Beispiel entspricht dem vorangehenden mit der Ausnahme, daß das CX-Register aus einer Speicherstelle geladen wird, deren Inhalt erst während der Programmausführung berechnet wird. Da der Schleifenzähler daher möglicherweise Null sein könnte, verwenden wir den Befehl JCXZ zur Überprüfung, ob die Schleife überhaupt durchlaufen werden soll.

```
        MOV     CX,LOOP_COUNT_WORD
        JCXZ    END_OF_LOOP
BEGIN_LOOP:
        ; . . . Programmschleife
        LOOP        BEGIN_LOOP
END_OF_LOOP:
```

Wir müssen den Befehl JCXZ durchaus nicht immer verwenden, wenn wir den Wert des Schleifenzählers erst im Programm berechnen. Wenn der Programmierer bereits weiß, daß der Wert Null nie auftreten kann, ist ein solcher Test unnötig. Allerdings lehrt uns die Erfahrung auch, daß gerade der Wert, der „nie" auftreten kann, im allgemeinen der erste ist, der bei der Programmausführung tatsächlich auftritt.

Die restlichen beiden LOOP-Befehle eröffnen uns noch weitergehende Möglichkeiten bei der Steuerung von Programmschleifen. Diese Befehle ähneln den Präfixen REP und REPNE. Während der LOOP-Befehl die Schleife nur dann verläßt, wenn der Inhalt von CX Null wird, tut dies der Befehl LOOPE (Loop while Equal) immer dann, wenn das Nullflag nicht gesetzt ist, oder CX den Wert Null enthält. Dies erlaubt einen doppelten Test auf die Schleifenendebedingung. Wir können also CX auf die maximale Anzahl der Schleidendurchläufe setzen und zugleich das Nullflag am Ende jedes Schleifendurchlaufs prüfen. Der Befehl LOOPNE (Loop while Not Equal) führt mit dem Nullflag den entgegengesetzten Test durch. Die Schleife wird also beendet, wenn CX Null enthält oder das Nullflag gesetzt ist.

Das folgende Beispiel zeigt die Verwendung des Befehls LOOPNE. Wir addieren dabei zwei parallele Zahlenreihen und wollen die Zahlenpaare herausfinden, deren Summe genau 100 ergibt. Da bei jedem Durchlauf die Zahlen vor dem Test addiert werden, können wir nicht einfach den Befehl REPNE CMPSB verwenden.

Wir setzen voraus, daß DS:SI und ES:DI so gesetzt sind, daß sie auf die beiden Zahlenreihen zeigen.

```
           MOV       CX,MAX_LOOP_COUNT  ; maximale Zahl der Durchläufe
BEGIN_LOOP:
           LODSB                        ; Zahl aus Reihe 1
           ADD       AL,ES:[DI]         ; addiere Zahl aus Reihe 2
           INC       DI                 ; nächste Zahl in Reihe 2
           CMP       AL,100             ; Summe = 100 ?
           LOOPNE    BEGIN_LOOP         ; weiter, falls kein Treffer und
                                        ; Ende noch nicht erreicht
           JE        MATCH_FOUND        ; Test auf Schleifenende bei
                                        ; Treffer
```

Prozessor-Steuerbefehle

Die verbleibenden Anweisungen aus dem Befehlssatz des 8088 steuern die Arbeit des Prozessors. Viele dieser Befehle setzen oder löschen die Anzeigen im Statusregister.

Setzen von Flags

Drei Befehle steuern direkt den Zustand des Carryflags. Dabei dienen STC, CLC und CMC zum Setzen, Löschen bzw. Komplementieren des Carryflags. Dieses Flag ist übrigens das einzige Flag des Bedingungscodes, mit dem dies möglich ist. Hauptgrund hierfür ist die Bedeutung des Carryflags in arithmetischen Operation von höherer Genauigkeit. Das Carryflag ist nämlich entscheidend für die Zwischenschritte bei arithmetischen Befehlen über mehrere Wörter. Die Möglichkeit zum Setzen bzw. Löschen dieses Flags kann also bei der dafür nötigen Schleifenbearbeitung hilfreich sein. In Abbildung 4.31 sehen wir ein Beispiel für die Verwendung des CLC-Befehls. Die dort enthaltene Schleife addiert die einzelnen Bytes von zwei zehnstelligen gepackten BCD-Zahlen. Die Routine wird fünfmal durchlaufen, da jedesmal zwei Ziffernstellen bearbeitet werden. Das Carryflag gibt dabei die Übertragsinformation von einem Schleifendurchlauf in den nächsten weiter. Mit CLC löschen wir das Carryflag vor dem ersten Schleifendurchlauf, um sicherzustellen, daß es keinen Übertrag in die erste Addition gibt. Auch für Rotationsbefehle ist das Carryflag von Bedeutung, da es hier während der Schiebeoperation zum neunten bzw. siebzehnten Bit des verwendeten Registers wird.

Auch für zwei Statusflags des Prozessors gibt es Befehle, die sie verändern können. In einem Programm können wir die Interruptmaske mit STI und CLI setzen bzw. löschen. Der Befehl STI dient zum Einschalten des Unterbrechungssystems des 8088 und ermöglicht die Annahme externer Unterbrechungen. Der Befehl CLI dagegen schaltet dieses System aus.

```
The IBM Personal Computer Assembler 01-01-83        PAGE    1-1
Figure 4.31  Multiple Precision BCD

1                                              PAGE    ,132
2                                              TITLE   Figure 4.31  Multiple Precision BCD
3
4       0000                            CODE   SEGMENT
5                                              ASSUME  CS:CODE, DS:CODE
6
7       = 0005                          NUMBER_LENGTH   EQU     5               ; 5 byte BCD numbers
8       0000      05 [                  NUMBER_ONE      DB      NUMBER_LENGTH DUP (?)
9                       ??
10                          ]
11
12      0005      05 [                  NUMBER_TWO      DB      NUMBER_LENGTH DUP (?)
13                      ??
14                          ]
15
16
17                                      ;--------------------------------
18                                      ; This routine adds the packed BCD number
19                                      ;  location NUMBER_ONE to the packed BCD number
20                                      ;  in location NUMBER_TWO, and leaves the result
21                                      ;  in NUMBER_TWO.
22                                      ;---------------------------------
23
24      000A                            START_ADD:
25      000A  B9 0005                          MOV     CX,NUMBER_LENGTH        ; Determine # bytes to add
26
27                                      ;----- Set index registers to point at least significant bytes
28
29      000D  BE 0004 R                        MOV     SI,OFFSET NUMBER_ONE + NUMBER_LENGTH - 1
30      0010  BF 0009 R                        MOV     DI,OFFSET NUMBER_TWO + NUMBER_LENGTH - 1
31
32      0013  F8                               CLC                             ; Start with no carry
33
34      0014                            ADD_LOOP:
35      0014  8A 04                            MOV     AL,[SI]                 ; Get ONE value
36      0016  02 05                            ADD     AL,[DI]                 ; Add the TWO value
37      0018  27                               DAA                             ; BCD adjust
38      0019  88 05                            MOV     [DI],AL                 ; Store the result
39      001B  9C                               PUSHF                           ; Save the carry flag
40      001C  4E                               DEC     SI                      ; Point to next byte of ONE
41      001D  4F                               DEC     DI                      ; Point to next byte of TWO
42      001E  9D                               POPF                            ; Get back the flags
43      001F  E2 F3                            LOOP    ADD_LOOP                ; Do next highest byte
44
45      0021                            CODE   ENDS
46                                             END
```

Abbildung 4.31 BCD-Rechnen mit höherer Genauigkeit

Auch das Richtungsflag können wir mit den dafür gedachten Befehlen STD (Set Direction) und CLD (Clear Direction) setzen bzw. löschen. Der Befehl CLD löscht das Richtungsflag und veranlaßt die Abarbeitung von Stringanweisungen in aufsteigender Richtung durch den Speicher. Der Befehl STD setzt dieses Flag und bedingt das Erniedrigen der Adresszeiger durch Stringoperationen.

Sonderbefehle

Der NOP-Befehl ist der schwächste im Befehlssatz des 8088. Er tut nämlich nichts — NO Operation. Eine genaue Untersuchung des generierten Maschinencodes enthüllt uns allerdings, daß er in der Tat ein XCHG-Befehl ist. Tatsächlich wird

XCHG AX,AX

ausgeführt, was, wie Sie sich vorstellen können, zu nichts führt. Obwohl der NOP-Befehl also nichts bewirkt, gibt es Fälle, wo seine Verwendung wünschenswert ist, beispielweise um Zeit zu gewinnen. Eine kurze Programmschleife, die aber eine bestimmte Zeit dauern soll, kann NOP-Anweisungen enthalten, um die gewünschte Durchlaufdauer zu erreichen (obwohl eine Schleife keineswegs die beste Methode ist, Zeit zu verbrauchen, außer es handelt sich um ein sehr kurzes Intervall). Die Ent-

wickler des IBM PC verwendeten NOPs an verschiedenen Stellen, um den Zeitbedarf einiger Hardwareteile auszugleichen. Beispielsweise kann ein Programm den Zeitgeber nicht öfter als einmal pro Mikrosekunde ansprechen. Zwei aufeinanderfolgende IN-Befehle verletzen aber diese Zeitbedingung, so daß wir hier mehrere NOP-Befehle zwischen die IN-Befehle einfügen müssen.

Der Befehl HLT (Halt) erfüllt genau diesen Zweck. Der Prozessor stoppt nach der Ausführung des Befehls. Sind zu diesem Zeitpunkt auch noch die Interrupts ausgeschaltet, so ist der Rechner tot. Die einzige Möglichkeit, ihn wieder zum Leben zu erwecken, ist, ihn aus- und wieder einzuschalten. Sind die Interrupts dagegen zum Zeitpunkt der Ausführung eines HLT-Befehls eingeschaltet, werden sie angenommen und die Steuerung geht an die Unterbrechungsroutine über. Nach der Rückkehr aus der Unterbrechungsroutine mit IRET wird das Programm an der unmittelbar auf HLT folgenden Stelle fortgesetzt. Wir können den HLT-Befehl in Mehrbenutzersystemen auch dazu verwenden, das gerade aktive Programm zu beenden, doch ist dies nur selten die beste Methode. Die Entwickler des PC haben die Verwendung des HLT-Befehls deshalb nur für den Fall eines schwerwiegenden Hardwarefehlers vorgesehen, wo eine weitere Programmausführung unklug erscheinen würde.

Der LOCK-Befehl ist eigentlich ein Präfix wie Segmentüberschreibungen oder das REP-Präfix. Er ist für Multiprozessorsysteme gedacht, bei denen mehrere Prozessoren auf die gleichen Speicherbereiche zugreifen könne. Das LOCK-Präfix veranlaßt den 8088, bestimmte Steuerleitungen anzusprechen, um damit exklusiven Zugriff auf den Speicher für die Dauer der mit dem Zusatz LOCK versehenen Anweisung zu erhalten. Das beste Beispiel hierfür ist das Setzen bzw. Testen eines Flags im gemeinsamen Speicherbereich.

```
MOV          AL,1
LOCK XCHG    AL,FLAG_BYTE
CMP          AL,1
```

In unserem Beispiel enthält FLAG_BYTE entweder 0 oder 1. Ein Prozessor setzt das Flag auf 1, wenn er einen geschützten Bereich des Programms erreicht, in dem er Systemoperationen durchführt, die immer nur ein Prozessor ausführen darf. Bevor er nun diesen Bereich betritt, muß er das Flag testen, um festzustellen, ob bereits ein anderer Prozessor darauf zugreift. Ist dem so, muß er warten; andernfalls kann er auf diesen Bereich zugreifen. Im Beispiel führen wir diesen Test durch und stellen damit sicher, daß kein anderer Prozessor auf den Bereich zugreift. Wir verwenden dafür den XCHG-Befehl mit LOCK-Präfix. LOCK ermöglicht dem Prozessor exklusiven Speicherzugriff während der Ausführung des XCHG-Befehls, der einem Lesen der gewünschten Speicherstelle, gefolgt von einem Schreiben auf dieselbe Stelle, entspricht. XCHG schreibt dabei die „1" aus dem AL-Register nach FLAG_BYTE, während er den aktuellen Inhalt dieser Stelle nach AL überträgt. Enthält AL nun „1", so greift bereits ein anderer Prozessor auf den geschützten Bereich zu, und wir müssen warten. Ist AL dagegen „0", so können wir den geschützten Bereich ansprechen, wobei XCHG die Stelle FLAG_BYTE auf „1" setzt, so daß wir dann von keinem anderen Prozessor mehr gestört werden können. Das LOCK-Präfix verhindert also, daß irgendein anderer Prozessor in der kurzen Zeitspanne zwischen Testen und Setzen des Flagbytes auf diesen Bereich zugreifen kann.

Allerdings ist die Erörterung des LOCK-Präfixes rein akademisch. Der IBM PC verfügt nämlich nicht über die nötige Hardware zur Ausführung des LOCK-Befehls.

Der WAIT-Befehl unterbricht ähnlich dem HLT-Befehl die Arbeit des Prozessors. Doch beim WAIT-Befehl wird die Arbeit wieder aufgenommen, wenn ein externer Eingang des 8088, der TEST-Eingang, aktiviert wird. Wird der WAIT-Befehl ausgeführt während der TEST-Eingang aktiviert ist, entsteht keine Verzögerung in der Ausführung. Ist der TEST-Eingang dagegen nicht aktiv, wartet der Prozessor, bis der Eingang aktiviert wird. Der 8088 verwendet diesen Befehl zusammen mit ESC, um den Zugang zum Arithmetikprozessor 8087 zu ermöglichen.

Der Befehl ESC (Escape) ermöglicht Erweiterungen im Befehlssatz des 8088, ohne den Prozessor verändern zu müssen. Der ESC-Befehl enthält ein Adressmodusfeld und kann eine beliebige Speicherstelle über die normalen Adressierungsarten des 8088 ansprechen. Allerdings liest der Prozessor nur den Inhalt dieser Speicherstelle, verwendet die Daten jedoch nicht.

Der Befehl ESC erlaubt es nun einem anderen Prozessor oder sogenannten Coprozessor, die Arbeit des 8088 zu beobachten. Die ESC-Anweisung aktiviert den Coprozessor, der sie dann selbst ausführt. Benötigt der Coprozessor eine Speicheradresse, so stellt sie der 8088 durch das scheinbare Lesen der Daten zur Verfügung. Der Coprozessor kann dann seinerseits die Daten an dieser Adresse in beliebiger Weise benützen. Die vollen Möglichkeiten dieses Befehls werden wir in Kapitel 7 erfahren, wo wir den Arithmetikprozessor 8087 als Coprozessor für den 8088 besprechen werden.

5 Über die Arbeit mit DOS und dem Assembler

Dieses Kapitel behandelt das notwendige Basistraining für das Assemblieren und die Ausführung von Programmen. In den vorangegangenen Kapiteln wurde die Arbeitsweise des 8088 dargestellt. Nun soll dieses Wissen in die Tat umgesetzt werden. Nur wenn Sie ein Programm tatsächlich erfolgreich schreiben und ausführen, können Sie wirklich mit dem Befehlssatz des 8088 umgehen.

In diesem Kapitel werden die vier Hauptphasen der Programmerstellung besprochen: Editieren, Assemblieren, Binden und Austesten. Jede dieser Phasen erfordert ein anderes Systemprogramm und ein anderes Verfahren. Alle diese Programme werden unter der Schirmherrschaft des Disk Operating Systems (DOS) ausgeführt. Drei der Programme gehören zum DOS-Paket, das Sie bei dem Händler erhalten können, der Ihnen auch Ihren IBM PC lieferte. Der Assembler wird getrennt davon verkauft, ist aber bei der gleichen Quelle erhältlich. Wir befassen uns hier übrigens ausschließlich mit IBM-Produkten.

Disk Operating System (DOS)

Wir werden mit dem DOS für die Programmerstellung und -ausführung arbeiten. Wir wollen deshalb zunächst erklären, was DOS ist und was es leisten kann. Das Disk Operating System bietet die Umgebung, in der andere Programme ausgeführt werden können. Bei Großrechnern ist das Betriebssystem ein Programm, das den Betrieb der Maschine steuert. Die meisten Großcomputer haben viele Benutzer, von denen jeder an die Betriebsmittel des Computers heran will. Das Betriebssystem stellt den Schiedsrichter dar, der darüber entscheidet, welcher Benutzer welche Betriebsmittel zu welcher Zeit benutzen darf. Das Betriebssystem verhindert dadurch, daß sich die verschiedenen Benutzer gegenseitig stören. Das Betriebssystem bietet außerdem Dienstleistungen an, die es dem Benutzer ersparen, sich mit den Problemen der Hardware befassen zu müssen. Diese Dienstprogramme verhindern, daß ein Benutzer die Hardware durch falsche Operationen so durcheinanderbringen kann, daß er die Daten und Programme anderer Benutzer zerstört. Die meisten Betriebssysteme von Großcomputern sind so konstruiert, daß ein Benutzerprogramm nicht direkt auf die Hardware einwirken kann. Die Bedürfnisse der vielen Benutzer beschränken also die individuellen Freiheiten der einzelnen Programme.

Bei kleinen Computern wie dem IBM PC dient das Betriebssystem einem anderen Zweck. Personalcomputer werden immer nur von einer Person gleichzeitig benutzt. Das Verwalten der Betriebsmittel des Computers erfolgt von außen her — wer an der Tastatur sitzt, hat alles beliebig in der Hand. Auf Programmebene können Anweisungen ungehindert eingegeben und ausgeführt werden. Das Programm kann nur denjenigen stören, der gerade mit dem Computer arbeitet und auch nur in der Form, daß das Programm oder eine Datei möglicherweise gelöscht werden.

Die Aufgabe des DOS für den IBM PC ist es, Benutzern eine betriebsfähige Umgebung sowie eine Anzahl von Dienstleistungen anzubieten. Der Benutzer kann ein

Programmierer, oder aber irgendein Anwendungsprogramm sein. Zum Beispiel, wenn Sie sich an die Tastatur setzen und mit dem System zu arbeiten beginnen, sind Sie ein DOS-Benutzer. Auch wenn der Editor eine Datei auf der Diskette oder Festplatte sichert, benutzt er DOS. Der Editor verwendet dabei Routinen von DOS, um eine Datei abzuspeichern und verfügt über keine eigenen Programmteile, die diese Aufgabe erledigen könnten.

Es ist die vornehmliche Aufgabe des DOS auf dem IBM PC, ein Dateisystem und die Ausführungsumgebung für Programme anzubieten. Mit Hilfe des Dateisystems können Daten auf einer Diskette oder Festplatte abgelegt und von dort zurückgespeichert werden. Wenn alle Anwendungsprogramme DOS benutzen, um Informationen zu speichern, dann sind diese Informationen auch allen zugänglich. Außerdem muß ein Programm dann auch nicht jedesmal ein eigenes Dateisystem beinhalten.

Dateisystem

Eine Diskette für den IBM PC kann 160 bis 360K Bytes Daten speichern. Eine Festplatte kann mehr als 10 Millionen Bytes speichern. Das Problem der Datenverwaltung ist also offensichtlich. Bei einer solchen Kapazität muß es einen Weg geben, alle Daten geordnet aufzubewahren. Als DOS-Benutzer möchten Sie eine bestimmte Menge von Daten als Einzeleinheit speichern, z.B. ein Assemblerprogramm. Es ist Ihnen dabei gleichgültig, wo auf der Diskette diese Daten gespeichert sind. Es ist Aufgabe des Systems, den Ort der gespeicherten Daten auf der Diskette herauszufinden.

Die Grundeinheit der Datenspeicherung ist eine Datei. Eine Datei ist eine Sammlung von Daten, welche in irgendeiner Form zusammengehören. Der Benutzer oder Ersteller einer Datei gibt ihr einen Namen. Alle Zugriffe auf die gespeicherten Daten erfolgen fortan über diesen Namen. Ein Programm, das auf diese gespeicherten Daten zugreift, muß folglich nicht mehr wissen, an welcher Stelle diese gespeichert sind. Eine Datei besteht aus Sätzen. Jeder Datensatz ist eine eigene Dateneinheit, aber nicht notwendigerweise ein Einzelbyte. Die einfachste Methode, Dateien und Datensätze zu verstehen, ist, sich zu vergegenwärtigen, was sie im Bürobereich bedeuten.

Eine Datei gleicht einem großen Schrank oder einer Akte mit vielen Unterlagen. Die Akte trägt gewöhnlich eine Bezeichnung — den Aktennamen. In der Akte befinden sich die jeweiligen Unterlagen. Die Akte eines Lehrers kann z. B. die Prüfungsunterlagen der Schüler enthalten, und jede Prüfung ist in dieser Akte sozusagen ein Datensatz. Der Lehrer hat die Unterlagen unter einer entsprechenden Beschriftung wie „1. Prüfung" gesammelt und abgelegt. Er findet die jeweiligen Prüfungsunterlagen, indem er zuerst nach der richtigen Akte greift und daraus dann den richtigen Datensatz entnimmt.

Wie hängt dies nun mit den Dateien eines Computers zusammen? Eine Datei ist eine Sammlung von miteinander in Beziehung stehenden Daten, und jede Datei hat einen Namen. Die Datensätze liegen in der Datei. Der Programmierer bestimmt die

Größe und den Inhalt der Datensätze. DOS steuert nicht das Format der Datensätze, es speichert lediglich die Datensätze in der Datei. DOS betrachtet einen Datensatz als eine Sammlung von Bytes in einer Datei. Der Programmierer bestimmt die Bedeutung der Bytes im Datensatz.

Wir wollen ein Assemblerprogramm als Beispiel für eine Datei betrachten. Das Programm hat einen Namen, und dieser Name wird der Dateiname. Die Datei setzt sich aus Datensätzen zusammen, wobei jeder Datensatz eine einzelne Assembleranweisung ist. Jeder Datensatz hat ein Format, das für den Assembler von Bedeutung ist, nicht jedoch für DOS. Die verschiedenen Bereiche in einem Datensatz sind Felder. DOS kümmert sich nicht darum, wie der Datensatz in Felder unterteilt ist. Das ist Aufgabe des Anwenderprogramms, in diesem Falle des Assemblers.

Dateinamen

Wenn wir über Dateinamen sprechen, sollten wir darauf eingehen, wie Sie bei DOS Dateien benennen können. Jeder DOS-Dateiname hat zwei Teile. Der erste Teil kann bis zu acht Zeichen lang sein. Dies ist der Teil, den der Benutzer spezifiziert, der „Name" der Datei. Der zweite Teil, bekannt als die Erweiterung, hat bis zu drei Zeichen. Das Anwendungsprogramm spezifiziert diesen Teil in der Regel als den Dateityp. Ein Punkt (".") trennt Name und Erweiterung voneinander. So hat z.B. COMMAND.COM als Namen COMMAND und als Erweiterung COM.

Manchmal kann der Benutzer die Erweiterung des Dateinamens selbst angeben. DOS oder ein Anwendungsprogramm benutzen die Dateinamenerweiterung jedoch für die Kennzeichnung des Dateityps. Bei COMMAND.COM kennzeichnet .COM eine Programmdatei. Der Assembler hat eine Eingabedatei und bis zu 3 Ausgabedateien. Die Eingabedateierweiterung ist .ASM für eine Assemblerdatei und die Ausgabeerweiterungen sind dementsprechend .OBJ, .LST und .CRF für Objekt-, Listen- und Querverweis-Dateien. In vielen Fällen erfordern die Anwendungsprogramme eine bestimmte Erweiterung der Dateinamen.

Inhaltsverzeichnis

Da eine Diskette sehr viele Informationen speichern kann, wäre es sehr verschwenderisch, eine ganze Diskette für eine einzelne Datei zu verwenden. Das Betriebssystem ermöglicht es uns, zahlreiche Dateien auf der Festplatte oder Diskette zu speichern.

Ein Speichermedium kann mehr als eine Datei aufnehmen, da DOS ein Inhaltsverzeichnis anlegt. Dieses Verzeichnis ist eine Tabelle des Inhalts. Es enthält alle Dateien auf der Platte bzw. Diskette. Außer dem Dateinamen legt DOS dabei auch noch weitere notwendige oder hilfreiche Information ab. Das Verzeichnis enthält die notwendigen Zeiger, um den eigentlichen Speicherplatz der Daten auf der Platte zu finden. Sehr praktisch ist im Inhaltsverzeichnis der Zeitstempel zu jeder Datei. Jedesmal, wenn ein Programm eine Datei anlegt oder aktualisiert, vermerkt DOS

Datum und Uhrzeit. Dies ist sehr hilfreich, wenn man es mit vielen Kopien einer Information zu tun hat und den jeweils neuesten Stand herausfinden muß.

Das Inhaltsverzeichnis löst das Problem von vielen auf einer einzigen Platte oder Diskette gespeicherter Dateien. DOS kann jedoch immer nur mit einem Laufwerk arbeiten. Hat Ihr System mehr als ein Platten- oder Diskettenlaufwerk, müssen Sie DOS mitteilen, in welchem Laufwerk die Datei untergebracht ist. Der Name des Diskettenlaufwerks wird dem Dateinamen als Präfix vorangestellt, z. B. hat COMMAND.COM im Laufwerk A als vollqualifizierten Namen A:COMMAND.COM.

Kommandoprozessor

Außer dem Dateisystem bietet DOS die Voraussetzung für den Ablauf von Anwenderprogrammen. Der erste Teil, mit dem der Benutzer in Berührung kommt, ist dabei der Kommandoprozessor. Dies ist der Teil von DOS, der die Befehle des Benutzers abarbeitet und die Ausführung von Anwendungsprogrammen einleitet.

Wenn Sie den IBM PC einschalten, so kann er praktisch nichts. Er mag zwar über eine große Leistungsfähigkeit verfügen, aber er kann nicht viel damit anfangen. Der Festspeicher (ROM) in der Maschine enthält Programme, die die Systemkomponenten testen (Power-On Self-Test, POST) und die Ein-/Ausgabegeräte initialisieren. Das restliche ROM Basic-Input/Output-System (BIOS) stellt eine Reihe von Dienstprogrammen zur Verfügung, die dem Assemblerprogrammierer helfen, auf Hardwareeinheiten zuzugreifen, ohne die Besonderheiten der Hardwareimplementierung berücksichtigen zu müssen. Das ist zwar alles recht und schön, aber es bietet nicht den Rahmen für die Ausführung von ernsthaften Anwendungsprogrammen.

DOS berücksichtigt dies. Nachdem POST das System initialisiert hat, lädt es DOS von der DOS-Diskette oder der Festplatte in den Arbeitsspeicher. Diesen Prozeß nennt man das „System booten". Der Ausdruck stammt von der englischen Redensart „lifting yourself by your own bootstraps" (sich selbst die Stiefel anziehen helfen). Die erste Aktion von DOS besteht darin, eine minimale Codemenge zu laden, die für das Laden des restlichen DOS benötigt wird. Damit baut sich DOS selbst auf. DOS beendet diese sogenannte Bootstrap-Prozedur mit der Ausgabe der Copyright-Meldung. In dieser Meldung ist unter anderem auch die aktuelle DOS-Version beinhaltet. Gelegentlich ist die DOS-Version wichtig, da jede neue DOS-Version zusätzliche Funktionen im System bedeutet.

Nach der Bootstrap-Operation kann DOS Befehle des Benutzers annehmen (bis auf einen Sonderfall, den wir später besprechen werden). Jetzt ist der Kommandoprozessor bereit. DOS hat den Kommandoprozessor, das Dateisystem und andere Dienstprogramme geladen, und diese sind nun betriebsbereit. Der Kommandoprozessor zeigt seine Bereitschaft an mit der Eingabeanforderung:



Dieses Symbol hat zwei Bedeutungen. Das Zeichen „>" signalisiert, daß der Kommandointerpreter auf einen Befehl wartet. Das „A" zeigt das voreingestellte Diskettenlaufwerk an. DOS kann nämlich Dateien immer nur diskettenweise verwalten.

Wenn mehr als eine Platte oder Diskette gleichzeitig benützt werden, muß der Benutzer angeben, welches Laufwerk DOS verwenden soll. Der IBM PC identifiziert die Laufwerke durch Buchstaben des Alphabets. Ein System mit zwei Diskettenlaufwerken hat ein A:Laufwerk und ein B:Laufwerk. Die Festplatte ist normalerweise das C:Laufwerk (die Laufwerke werden in der Regel mit einem nachgestellten Doppelpunkt angegeben). Das voreingestellte Laufwerk enthält die Dateien, mit denen DOS arbeitet, es sei denn der Benutzer gibt eine anderslautende Anweisung. Um eine Datei auf einem voreingestellten Laufwerk ansprechen zu können, benötigt DOS lediglich den Dateinamen. Soll eine Datei auf irgendeinem anderen Laufwerk angesprochen werden, muß DOS diesen Laufwerknamen und den Dateinamen wissen.

DOS wird nur auf Kommandos des Benutzers hin tätig. Alle DOS-Kommandos benötigen als Input eine Antwort auf die Eingabeanforderung „>". Der Benutzer gibt nun den Kommandonamen ein, worauf der Kommandointerpreter diese Eingabe bearbeitet. Wie der Interpreter die Eingabe handhabt, hängt vom eingegebenen Kommando ab. Es gibt residente Kommandos, die immer zur Verfügung stehen, und Kommandos, die eine nichtresidente bzw. transiente Operation aufrufen. Diese Kommandos benötigen eine spezielle Diskettendatei, bevor sie ausgeführt werden können.

Die residenten Kommandos unterstützen das Dateisystem. Sie sind im DOS fest gespeichert, da sie für die Verwaltung der Diskettendaten häufig gebraucht werden. Bei der Benutzereingabe übergibt der Kommandointerpreter die Steuerung an die entsprechende Routine im DOS. Diese führt die gewünschte Funktion den Spezifikationen des Kommandos entsprechend aus und übergibt die Steuerung wieder an DOS. Abbildung 5.1 zeigt die residenten Kommandos von DOS.

Befehl	Aktion
COPY	Kopieren einer Datei
DATE	Ausgeben oder Ändern des aktuellen Datums
DIR	Ausgeben des Inhaltsverzeichnisses einer Diskette
ERASE	Löschen einer Datei von einer Diskette
PAUSE	Warten, bis ein Zeichen von der Tastatur kommt
REM	Kommentar
RENAME	Ändern des Namens einer Diskettendatei
TIME	Ausgeben oder Ändern der aktuellen Uhrzeit
TYPE	Ausgeben des Inhalts einer Datei

Abbildung 5.1 Residente DOS-Kommandos

Ein typisches residentes Kommando ist DIR. Dieses Kommando gibt das Inhaltsverzeichnis einer Diskette aus. Abbildung 5.2 zeigt die Ausgabe des DIR-Kommandos. Beachten Sie, daß das DIR-Kommando Name und Erweiterung aller Dateien auf der Diskette zeigt. Es werden dabei die Länge in Bytes sowie das Datum und die Uhrzeit der Erstellung angegeben. Da DOS das Inhaltsverzeichnis einer Diskette z.B. auch lesen muß, um ein Programm zu laden, ist das DIR-Kommando eine residente Funktion.

```
A>dir
COMMAND    COM     4959   4-06-82   12:00p
FORMAT     COM     3816   4-06-82   12:00p
CHKDSK     COM     1720   4-06-82   12:00p
SYS        COM      605   4-06-82   12:00p
DISKCOPY   COM     2008   4-06-82   12:00p
DISKCOMP   COM     1640   4-06-82   12:00p
COMP       COM     1649   4-06-82   12:00p
EXE2BIN    EXE     1280   4-06-82   12:00p
MODE       COM     2509   4-06-82   12:00p
EDLIN      COM     2392   4-06-82   12;00p
DEBUG      COM     5999   4-06-82   12:00p
LINK       EXE    41856   4-06-82   12:00p
BASIC      COM    11136   1-01-80    4:43a
BASICA     COM    16512   1-01-80    4:55a
ART        BAS     1920   4-06-82   12:00p
SAMPLES    BAS     2432   4-06-82   12:00p
MORTGAGE   BAS     6272   4-06-82   12:00p
COLORBAR   BAS     1536   4-06-82   12:00p
CALENDAR   BAS     3840   4-06-82   12:00p
MUSIC      BAS     8704   4-06-82   12:00p
DONKEY     BAS     3584   4-06-82   12:00p
CIRCLE     BAS     1664   4-06-82   12:00p
PIECHART   BAS     2304   4-06-82   12:00p
SPACE      BAS     1920   4-06-82   12:00p
BALL       BAS     2048   4-06-82   12:00p
COMM       BAS     4352   4-06-82   12:00p
        26 File(s)
```

Abbildung 5.2 Disketten-Inhaltsverzeichnis

Gibt der Benutzer ein nichtresidentes Kommando ein, versucht der Kommandointerpreter das gewünschte Kommando von der Platte oder Diskette zu laden. In diesem Falle dient der Interpreter als Programmlader. Der Interpreter geht dabei davon aus, daß der Kommandoname mit dem Dateinamen identisch ist. Er durchsucht das Inhaltsverzeichnis nach einer Datei, deren Name mit dem des Kommandos übereinstimmt und lädt diese Datei. Der Interpreter übergibt anschließend die Steuerung an dieses Programm, so daß dieses dann seine eigentliche Aufgabe, den angegebenen Befehl, ausführen kann.

Der Kommandoprozessor lädt nicht einfach irgendeine Datei. Die Dateinamenerweiterung muß entweder .COM oder .EXE sein. Dies sind entsprechende Programm- und Ausführungsdateien. Das Endprodukt der Assemblier- und Linkoperation ist eine .EXE-Datei. Das bedeutet, daß Sie Ihr eigenes DOS-Kommando schreiben können. Wenn Sie ein Assemblerprogramm geschrieben, assembliert und gebunden haben und dieses Programm auf der Diskette belassen, können Sie es wie jedes andere DOS-Programm laden und ausführen.

Es besteht ein Unterschied zwischen .COM- und .EXE-Dateien. Sie haben etwas unterschiedliche Strukturen und werden verschieden gesteuert. Obgleich .EXE-

Dateien das normale Endprodukt des Bindens sind, gibt es doch gelegentlich Gründe dafür, mit .COM-Dateien zu arbeiten. In einem späteren Abschnitt werden wir die Unterschiede zwischen diesen Dateien behandeln und erläutern, wie man eine .EXE-Datei in eine .COM-Datei umwandeln kann.

Lassen Sie uns nun das Beispiel eines transienten Kommandos betrachten. Der Makroassembler ist dazu ein gutes Beispiel. Um den Assembler aufzurufen, geben Sie das Kommando

```
A>ASM
```

ein. Wenn Sie das Inhaltsverzeichnis der Assembler-Diskette durchsehen, werden Sie feststellen, daß es dort eine Datei mit dem Namen ASM.EXE gibt. Diese Datei ist der Assembler. Aufgrund des ASM-Kommandos sucht der Kommandoprozessor auf der Diskette im Laufwerk A: (die voreingestellte Diskette). Findet er die Datei ASM.EXE, lädt er den Assembler und übergibt ihm die Steuerung. Der Assembler steuert nun das System. Er wird – so ist zu hoffen – die Steuerung an den Kommandoprozessor zurückgeben, wenn er das Assemblieren beendet hat. Beachten Sie, daß die Assemblerdatei eine .EXE-Datei ist, so daß der Kommandoprozessor sie laden kann.

Befindet sich der Assembler nicht auf der Diskette im Laufwerk A:, kann der Benutzer das Laufwerk mit

```
A>B:ASM
```

spezifizieren. Das Präfix B: sagt DOS, daß die Datei auf der Diskette im Laufwerk B: ist. Eine Datei ist vollständig spezifiziert, wenn nicht nur der Dateiname, sondern auch das Laufwerk, auf welchem sie residiert, angegeben ist. Der Dateiname alleine genügt nur für eine Datei auf dem voreingestellten Laufwerk. Um eine Datei im Laufwerk B: mit dem Assembler im Laufwerk A: zu assemblieren, gilt folgendes Kommando:

```
A>ASM  B:FILE.ASM
```

Dieses Kommando spezifiziert beide Dateien, das Programm ASM im voreingestellten Laufwerk und die Quelldatei FILE.ASM im Laufwerk B:.

Sie können auch den anderen Weg gehen. Sie können das Laufwerk B: zum voreingestellten Laufwerk machen und das Kommando wie folgt spezifizieren:

```
A>B:
B>A:ASM  FILE.ASM
```

Das Kommando B: veranlaßt DOS auf B: als voreingestelltes Laufwerk umzuschalten. Beachten Sie, daß die Eingabeanforderung auf B> übergewechselt hat und anzeigt, daß B: das voreingestellte Laufwerk ist. Bei diesem Beispiel ist das Kommando mit dem vorgenannten identisch.

Der Kommandointerpreter kann auch eine Datei mit einer Erweiterung von .BAT ausführen, die Stapelverarbeitungsdatei bzw. Batch-Datei genannt wird. Dieser Dateityp ist ein ganz anderer als die .COM- oder .EXE-Dateien. Eine .BAT-Datei enthält keinen Maschinencode für die Ausführung. Die Batchdatei beinhaltet Kommandos,

die der Kommandoprozessor interpretiert. Die .BAT-Datei enthält DOS-Kommandos, welche der Reihe nach ausgeführt werden. Man kann eine .BAT-Datei als Ersatz für die Eingabe über die Tastatur betrachten. Anstatt Kommandos über die Tastatur einzugeben, stapelt diese Datei die Kommandos. Sobald DOS den Inhalt der .BAT-Datei ausgeführt hat, kehrt es zur Tastatur zurück, um das nächste Kommando anzufordern. So kann man mit der Batchdatei sich wiederholende Aufgaben auf sehr praktische Weise erledigen. Wenn die Batchdatei einmal erstellt ist, kann man so mit einem einzigen Kommando alle anderen Kommandoeingaben ersetzen.

Es gibt eine spezielle Batchdatei mit dem Namen AUTOEXEC.BAT. DOS führt diese Datei, sofern sie vorhanden ist, unmittelbar nach Systemstart aus. DOS übergibt die Steuerung unverzüglich an die Kommandos der Batchdatei. Dadurch geht die Diskette automatisch zu einem gewünschten Benutzerprogramm. Nehmen wir an, Sie hätten ein Anwendungsprogramm geschrieben, das DOS-Funktionen benutzt. (In diesem Falle sagen wir in der Regel, daß das Programm geschrieben wurde, um „unter DOS zu laufen.") Durch die Erstellung einer AUTOEXEC.BAT-Datei, die das Anwendungsprogramm startet, braucht der Anwender den Umgang mit dem Kommandointerpreter nicht zu lernen. Das erste, was ein solcher Benutzer erfährt, ist, daß das für ihn gedachte Programm läuft.

DOS-Funktionen

Der Kommandointerpreter bietet einen Mechanismus, mit dem Ihr Assemblerprogramm zur Ausführung gebracht wird. Wenn es läuft, bietet DOS durch die DOS-Funktionen außerdem Zugang zum Dateisystem. Dieser Abschnitt zeigt, welcher Art diese Funktionen sind und wie ein Programm sie benutzen kann. Anhang D des Disk Operating System Reference Manual enthält eine vollständige Beschreibung dieser Funktionen.

Ein Programm führt eine DOS-Funktion über einen Softwareinterrupt aus. Mit Hilfe einer Unterbrechung kann ein Programm eine Routine aufrufen, ohne zu wissen, wo diese sich wirklich befindet. Der Programmierer spezifiziert dazu nur die gewünschte Unterbrechung. Bei der DOS-Initialisierung werden die Unterbrechungsvektoren für die DOS-Funktionen gesetzt, die dann auf die richtigen Routinen zeigen. Auf diese Weise erübrigt sich die Modifizierung Ihres Programms, obwohl verschiedene DOS-Versionen in Umlauf sind. Abbildung 5.3 zeigt die DOS-Unterbrechungsvektoren.

Interrupt	Operation
20H	Programmende
21H	Funktionsaufruf (siehe Abbildung 5.5)
22H	Endadresse
23H	CTRL-BREAK Adresse
24H	Fehlerbehandlung
25H	absoluter Plattenlesebefehl
26H	absoluter Plattenschreibbefehl
27H	Programmende ohne Speicherfreigabe

Abbildung 5.3 DOS-Interrupts

Einige dieser Unterbrechungen sind eigentlich für Benutzerprogramme gedacht. Die Interrupts 22H, 23H und 24H sind Pointer auf Routinen, die in Ihrem Programm enthalten sein können. Diese Vektoren geben das auszuführende Programm an, wenn der dazugehörige Zustand auftritt. So führt DOS z.B. die Unterbrechung 23H aus, wenn die Taste CTRL-BREAK gedrückt wird. Dieser Tastendruck signalisiert normalerweise eine Unterbrechung in der Programmausführung. DOS besetzt diese drei Unterbrechungen mit der normalen Unterbrechungsbehandlung vor. Wenn es für ein Programm notwendig ist, diese Unterbrechungen anders zu behandeln, kann es den Interruptvektor ändern.

Lassen Sie uns z. B. Interrupt 24H betrachten, den normalen Fehlerbearbeiter. Immer wenn DOS einen Fehler entdeckt, ruft es diese Unterbrechung auf. Normalerweise zeigt dieser Vektor auf eine DOS-Routine, die eine Fehlermeldung auf dem Bildschirm ausgibt. Für einen Diskettenfehler zeigt DOS also eine Diskettenfehlermeldung. Der Benutzer hat die Wahl, die fehlerverursachende Operation zu wiederholen, abzubrechen oder zu ignorieren. Für unser Beispiel wollen wir annehmen, daß Sie ein Programm schreiben, um Disketten zu formatieren. Diese Operation legt physikalisch Spuren und Sektoren auf der Diskette fest. Als Teil des Formatierens werden Sie die Diskette auf fehlerhafte Bereiche testen, bevor Sie sie benutzen. Dies nennt man „Prüfen" einer Diskette. Da es möglich ist, daß eine Diskette ein oder zwei defekte Stellen hat und sonst brauchbar ist, werden Sie diese schlechten Stellen besonders kennzeichnen, so daß sie später nicht versehentlich benutzt werden.

Das Formatierungsprogramm ersetzt hier die Fehlerbehandlung. Sie wollen nämlich nicht, daß DOS eine Fehlermeldung für den Benutzer ausgibt; vielmehr möchten Sie, daß das Programm den festgestellten Fehler beachtet und diesen als defekte Stelle auf der Diskette markiert. Um dies zu erreichen, ersetzen Sie den Unterbrechungsvektor an der Stelle 00090H (24H × 4) durch einen Zeiger auf das fehlerbearbeitende Programm. Wenn DOS nun einen Diskettenfehler entdeckt, kann Ihr Programm den Ort des Fehlers als defekte Stelle markieren ohne den Benutzer von dem Problem besonders unterrichten zu müssen.

Die Unterbrechungen 25H und 26H verbinden zwei Teile von DOS miteinander. Die Dateibehandlung von DOS besteht tatsächlich aus zwei Teilen. Der eine Teil, IBMBIO.COM, greift direkt auf die Hardware zu, der andere, IBMDOS.COM, stellt die Dateiverwaltung dar. Wenn das Dateisystem von DOS nun mit dem BIO (Basic I/O) in Verbindung tritt, benutzt es diese beiden Unterbrechungen. Wenngleich diese Interrupts auch Ihnen zur Verfügung stehen, besteht ihr vornehmlicher Zweck doch in der Trennung der beiden genannten Teile von DOS. Unterbrechung 25H bedeutet dabei absolutes Plattenlesen, während Unterbrechung 26H absolutes Plattenschreiben bedeutet. Auf dieser untersten Ebene arbeitet das Interface mit absoluten Positionen auf der Diskette und nicht mehr mit Datensätzen innerhalb einer Datei.

Die Unterbrechungen 20H und 27H bieten einen Mechanismus, mit dem nach Ausführung eines Programmes die Steuerung an DOS zurückgegeben wird. Unterbrechung 20H ist dabei die normale Rückkehr aus einem Programm. Unterbrechung 27H ist insofern interessant, als sie das Programm beendet, den vom Programm benutzten Speicher aber nicht an DOS zurückgibt. Was im Speicher belassen wird, bleibt dort, bis der Rechner abgestellt oder das System neu gestartet wird. Diese

Funktion ist wertvoll, wenn Sie eine neue Unterbrechungsroutine oder eine ähnliche Funktion, die Teil des Systems werden soll, bereitstellen wollen. In Kapitel 10 werden wir ein Beispiel zeigen, das den INT 27H-Ausgang benutzt, um das System zu erweitern.

Eine Warnung für die Benutzung der Unterbrechungen 20H und 27H: Sie sollten nur aus einer .COM-Datei ausgeführt werden. Der Unterschied zum Format einer .EXE-Datei genügt bereits, um diese DOS-Funktionen etwas schwieriger zu gestalten. Im

Wert in AH	Funktion
0	Programmende
1	Tastatureingabe
2	Bildschirmausgabe
3	zusätzl. Eingabe (asynchrone Schnittstelle)
4	zusätzl. Ausgabe
5	Druckerausgabe
6	direkte Bildschirm/Tastatur I/O
7	direkte Tastatureingabe mit Echo
8	Tastatureingabe ohne Echo
9	String ausgeben
0AH	gepufferte Tastatureingabe
0BH	Prüfen Tastaturstatus
0CH	Löschen Tastaturpuffer und Ausführen der Funktion in AL
0DH	Disk Reset
0EH	Select Disk
0FH	Datei eröffnen
10H	Datei schließen
11H	Suchen ersten Eintrag
12H	Suchen nächsten Eintrag
13H	Datei löschen
14H	Lesen sequentiell
15H	Schreiben sequentiell
16H	Erzeugen Datei
17H	Datei umbenennen
19H	aktuelles Laufwerk
1AH	Setzen Disk Transfer Adresse (DTA)
1BH	Adresse der Dateizuordnungstabelle
21H	Lesen direkt
22H	Schreiben direkt
23H	Dateigröße
24H	Setzen Random Record Field
25H	Setzen Interruptvektor
26H	Erzeugen neues Programmsegment
27H	Lesen direkt Block
28H	Schreiben direkt Block
29H	Suchen Datei über Dateiname
2AH	Holen Datum
2BH	Setzen Datum
2CH	Holen Zeit
2DH	Setzen Zeit

Abbildung 5.4 Funktionen des DOS-Interrupts 21H

nächsten Abschnitt werden wir die Unterschiede zwischen .COM- und .EXE-Dateien besprechen und den Grund dafür erörtern, warum die Interrupts 20H und 27H jeweils verschieden damit arbeiten.

Interrupt 21H ist die Unterbrechung zum Aufruf einer DOS-Funktion. Diese Unterbrechung ermöglicht den Zugriff auf die DOS I/O-Struktur. Abbildung 5.4 zeigt die Funktionen, die über diese Unterbrechung aufrufbar sind. Ein Programm wählt eine Funktion aus, indem es das AH-Register auf den gewünschten Wert setzt und dann die Unterbrechung 21H erzeugt.

Die Parameter für diese Funktionen sind im Anhang D des DOS Manual aufgeführt. Wir wollen lieber ein Beispiel durchgehen, das einige dieser Funktionen benutzt, als jede Funktion im Detail besprechen. Das Beispiel berücksichtigt speziell die DOS-Diskettenfunktionen.

Dateisteuerblock

Bevor wir das Beispiel durchgehen können, müssen wir uns mit einer Datenstruktur des DOS beschäftigen, die ein wesentlicher Teil des Dateisystems ist — dem Dateisteuerblock (FCB). Diese Struktur wird für alle Dateioperationen benutzt.

Der Dateisteuerblock ist die Schnittstelle zwischen Benutzerprogramm und DOS-Funktionen. Jede Dateiaktion verweist auf den FCB. Abbildung 5.5 zeigt den Aufbau des Standard-FCB. Es gibt eine Variation des FCB, bekannt als „erweiterter FCB", der in bestimmten Situationen benutzt wird. Er wird verwendet, um eine Datei zu „verstecken". Eine versteckte Datei hat eine Schreibsperre. Dies bedeutet, daß ein Programm die Datei nicht modifizieren kann, ohne zuvor den FCB für die Datei zu ändern. Eine versteckte Datei erscheint nicht bei der Ausgabe des Inhaltsverzeichnisses. Das Verstecken einer Datei ist eine einfache Methode, sie vor einem ungeschickten Benutzer zu schützen. Unsere folgenden Beispiele beschäftigen sich jedoch ausschließlich mit dem Standard-FCB.

Die Felder des FCB beinhalten auch die Dateiattribute. Laufwerknummer, Dateiname und Erweiterung ergeben die Kennzeichnung für die Datei. Dateigröße und Datum sind Dateiattribute, die wir bereits im Inhaltsverzeichnis gesehen haben. Die verbleibenden Felder — aktuelle Blocknummer, Satzlänge, relative Satznummer und absolute Satznummer — positionieren die Datei für Lesen und Schreiben. Die Satzlänge bestimmt den Umfang des benutzerdefinierten Datensatzes in Bytes. Alle Schreib- und Lesebefehle für eine Datei werden auf Datensatzebene ausgeführt. Die Satzlänge bestimmt dabei die Menge der Daten, die bei jeder Operation transportiert werden.

Es gibt zwei Methoden, um eine Datei für den Zugriff auf Datensätze einzustellen. Die erste Methode, die sequentielle, behandelt die Datensätze der Reihe nach. Die laufende Blocknummer und die relative Datensatznummer identifizieren den näch-

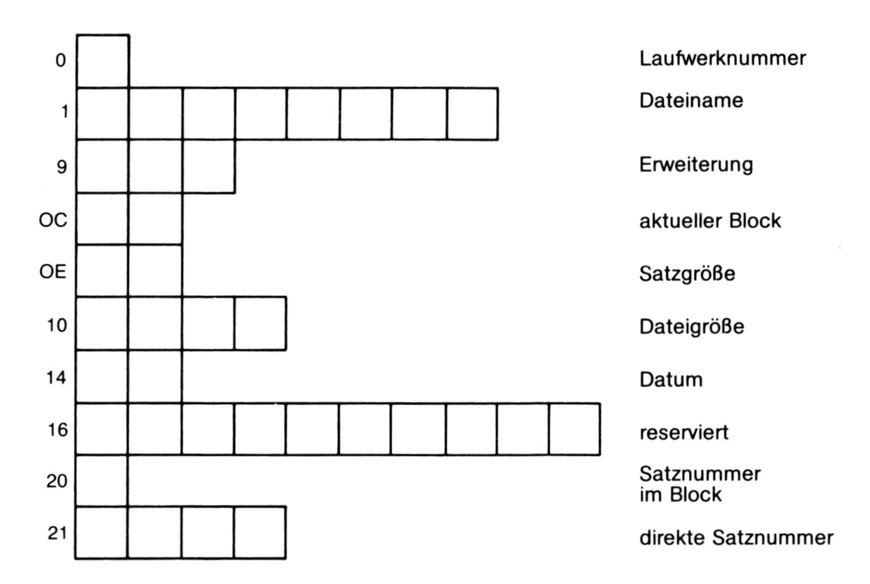


Abbildung 5.5 Dateisteuerblock

sten Datensatz, auf den zugegriffen werden soll. Während das Programm eine Lese- oder Schreiboperation durchführt, erhöht DOS die relative Datensatznummer, um auf den nächsten Datensatz zu deuten. Sequentielle Operationen durchlaufen die Datei vom Beginn bis zum Ende. Eine Assembler-Quelldatei ist ein Beispiel für eine sequentielle Datei. Der Assembler liest Datensätze vom Beginn bis zum Ende der Datei.

Eine Datei kann für ein Programm auch als Direktzugriffsdatei zugänglich sein. Das englische „random" (= direkt oder wahlfrei) bedeutet nicht, daß ein Datensatz nach Zufallskriterien ausgewählt wird. Es bedeutet vielmehr, daß das Programm einen beliebigen Datensatz in der Datei als nächsten auswählen kann. Ein anderer Ausdruck für „random file" ist deshalb auch „direct file". Das Programm kann auf jeden Datensatz direkt zugreifen.

Ein Programm kann mit einer Datei auf beide Arten arbeiten. Für sequentielle Operationen aktualisiert DOS automatisch die relative Satznummer im FCB. Das Programm muß jedoch selbst eine Satznummer für den Direktzugriff auf die Datei liefern. Eine Direktzugriffsdatei ist eine Dateistruktur, die wir mit einem Vektor vergleichen können. Genau wie das Programm einen Index liefern muß, um ein Element in einem Vektor zu finden, muß das Programm eine Satznummer liefern, um einen Datensatz in einer Direktzugriffsdatei aufzugreifen.

Wir wollen uns nun diese DOS-Funktionen und den FCB in einem Beispiel ansehen. Abbildung 5.6 ist ein langes Programm, das wenig tut. Der Zweck des Programmes ist, die Funktionen der Unterbrechung 21H des DOS sowie den Gebrauch des FCB aufzuzeigen. Die in diesem Programm durchgeführten Operationen können aber auch für ein wirkliches Problem verwendet werden.

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 5.6  DOS Function Example
1                                                 PAGE    ,132
2                                                 TITLE   Figure 5.6  DOS Function Example
3        0000                             CODE    SEGMENT
4        005C                                     ORG     05CH                            ; First FCB location
5        005C                             FCB             LABEL   BYTE                    ; Block label
6        005C  ??                         FCB_DRIVE               DB      ?               ; Drive number
7        005D      08 [                   FCB_NAME                DB      8 DUP(?)        ; Name
8                         ??
9                              ]
10
11       0065      03 [                   FCB_EXT                 DB      3 DUP(?)        ; Extension
12                        ??
13                             ]
14
15       0068  ????                       FCB_BLOCK               DW      ?               ; Current block number
16       006A  ????                       FCB_RECORD_SIZE         DW      ?               ; Record size
17       006C  ????????                   FCB_FILE_SIZE           DD      ?               ; Total size of file
18       0070  ????                       FCB_DATE                DW      ?               ; Date created
19       0072      0A [                   FCB_RESV                DB      10 DUP(?)       ; Reserved for DOS use
20                        ??
21                             ]
22
23       007C  ??                         FCB_CURRENT_RECORD  DB          ?               ; Current record number
24       007D  ????????                   FCB_RANDOM_RECORD   DD          ?               ; Random record number
25       0090                                     ORG     090H
26       0090                             DISK_TRANSFER_ADDRESS   LABEL   BYTE            ; Data buffer area
27       0100                                     ORG     100H
28                                                ASSUME  CS:CODE, DS:CODE, ES:CODE
29       0100  E9 01B8 R                          JMP     PROGRAM_START                   ; Jump around messages
30
31       = 0020                           RECORD_SIZE     EQU     32                      ; Size of records
32       0103  03 00 00 00 00             KEYBOARD_BUFFER DB      3,0,0,0,0               ; For buffered input
33       0108  46 69 6C 65 20 61          FILE_ERROR_MSG  DB      'File already exists',10,13,'$'
34             6C 72 65 61 64 79
35             20 65 78 69 73 74
36             73 0A 0D 24
37       011E  55 6E 61 62 6C 65          BAD_OPEN_MSG    DB      'Unable to open file',10,13,'$'
38             20 74 6F 20 6F 70
39             65 6E 20 66 69 6C
40             65 0A 0D 24
41       0134  45 72 72 6F 72 20          BAD_WRITE_MSG   DB      'Error in writing to file',10,13,'$'
42             69 6E 20 77 72 69
43             74 69 6E 67 20 74
44             6F 20 66 69 6C 65
45             0A 0D 24
46       014F  45 72 72 6F 72 20          BAD_READ_MSG    DB      'Error in reading file',10,13,'$'
47             69 6E 20 72 65 61
48             64 69 6E 67 20 66
49             69 6C 65 0A 0D 24
50       0167  45 72 72 6F 72 20          BAD_CLOSE_MSG   DB      'Error in closing file',10,13,'$'
51             69 6E 20 63 6C 6F
52             73 69 6E 67 20 66
53             69 6C 65 0A 0D 24
54       017F  54 77 6F 20 63 68          INPUT_BAD_MSG   DB      'Two character input required',10,13,'$'
55             61 72 61 63 74 65
56             72 20 69 6E 70 75
57             74 20 72 65 71 75
58             69 72 65 64 0A 0D
59             24
60       019E  45 6E 74 72 79 20          CHAR_BAD_MSG    DB      'Entry must be char|char',10,13,'$'
61             6D 75 73 74 20 62
62             65 20 63 68 61 72
63             7C 63 68 61 72 0A
64             0D 24
65
66                                        ;----- Set Disk Transfer Address
67
68       01B8                             PROGRAM_START:
69       01B8  B4 1A                              MOV     AH,1AH                          ; Set disk transfer address
70       01BA  BA 0090 R                          MOV     DX,OFFSET DISK_TRANSFER_ADDRESS
71       01BD  CD 21                              INT     21H
72
73                                        ;----- Search for similar file
74
75       01BF  B4 11                              MOV     AH,11H                          ; Do a search for entry
76       01C1  BA 005C R                          MOV     DX,OFFSET FCB                   ;  to see if a file with
77       01C4  CD 21                              INT     21H                             ;  this name already exists
78       01C6  0A C0                              OR      AL,AL
79       01C8  75 09                              JNZ     NO_FILE                         ; Jump if this is new name
80       01CA  BA 0108 R                          MOV     DX,OFFSET FILE_ERROR_MSG        ; Identical file error msg
81       01CD                             ERROR_EXIT:
82       01CD  B4 09                              MOV     AH,9                            ; This routine will display
83       01CF  CD 21                              INT     21H                             ;  the string and exit
84       01D1  CD 20                              INT     20H                             ;  from the program
85
86                                        ;----- Create the file
87
88       01D3                             NO_FILE:
89       01D3  B4 16                              MOV     AH,16H                          ; Create the file
90       01D5  BA 005C R                          MOV     DX,OFFSET FCB
91       01D8  CD 21                              INT     21H
92       01DA  0A C0                              OR      AL,AL                           ; Test for success
93       01DC  74 05                              JZ      CREATE_OK
94       01DE  BA 011E R                          MOV     DX,OFFSET BAD_OPEN_MSG          ; Create failed error msg
95       01E1  EB EA                              JMP     ERROR_EXIT
96
97                                        ;----- Set up the FCB parameters
98
99       01E3                             CREATE_OK:
100      01E3  C6 06 007C R 00                    MOV     FCB_CURRENT_RECORD,0            ; Initialize current record
```

```
101   01E8  C7 06 007D R 0000           MOV    WORD PTR FCB_RANDOM_RECORD,0     ;   and random record fields
102   01EE  C7 06 007F R 0000           MOV    WORD PTR FCB_RANDOM_RECORD+2,0
103   01F4  C7 06 006A R 0020           MOV    WORD PTR FCB_RECORD_SIZE,RECORD_SIZE    ; Set record size
104
105                               ;----- Write out the original file
106                                                                          ; This loop will write
107   01FA  B0 41                       MOV    AL,'A'                        ;  out the letters
108   01FC                        CHARACTER_LOOP:                            ;  of the alphabet
109   01FC  BF 0090 R                   MOV    DI,OFFSET DISK_TRANSFER_ADDRESS
110   01FF  B9 0020                     MOV    CX,RECORD_SIZE
111   0202  F3/ AA                      REP    STOSB                         ; Fill data area with chars
112   0204  50                          PUSH   AX                            ; Save the character
113   0205  BA 005C R                   MOV    DX,OFFSET FCB
114   0208  B4 15                       MOV    AH,15H                        ; Sequential write of this
115   020A  CD 21                       INT    21H                           ;  character block
116   020C  0A C0                       OR     AL,AL                         ; Test for successful write
117   020E  58                          POP    AX                            ; Recover character
118   020F  74 05                       JZ     WRITE_OK
119   0211  BA 0134 R                   MOV    DX,OFFSET BAD_WRITE_MSG       ; Write failed error msg
120   0214  EB B7                       JMP    ERROR_EXIT
121   0216                        WRITE_OK:
122   0216  FE C0                       INC    AL                            ; Move to next character
123   0218  3C 5B                       CMP    AL,'Z'+1                      ; Wrote last character?
124   021A  75 E0                       JNE    CHARACTER_LOOP                ; Write the next character
125
126                               ;-----Inquiry/update section
127
128   021C                        KEYBOARD_LOOP:                             ; This loop will accept
129   021C  BA 0103 R                   MOV    DX,OFFSET KEYBOARD_BUFFER     ;  input and modify the
130   021F  B4 0A                       MOV    AH,0AH                        ;  file
131   0221  CD 21                       INT    21H                           ; Buffered keyboard input
132   0223  80 3E 0104 R 02             CMP    KEYBOARD_BUFFER+1,2           ; Were two characters read?
133   0228  74 09                       JE     KEY_INPUT_OK
134   022A  BA 017F R                   MOV    DX,OFFSET INPUT_BAD_MSG       ; Input error msg
135   022D                        KEYBOARD_ERROR:
136   022D  B4 09                       MOV    AH,9
137   022F  CD 21                       INT    21H                           ; Print bad input message
138   0231  EB E9                       JMP    KEYBOARD_LOOP                 ; Go back for more kb input
139   0233                        KEY_INPUT_OK:
140   0233  BA 019E R                   MOV    DX,OFFSET CHAR_BAD_MSG        ; Set up pointer in case
141   0236  A0 0105 R                   MOV    AL,KEYBOARD_BUFFER+2          ; Get first keystroke
142   0239  3C 24                       CMP    AL,'$'                        ; Test for exit character
143   023B  75 03                       JNE    CHANGE_RECORD                 ; Change record if not
144   023D  EB 55 90                    JMP    PROGRAM_EXIT
145
146                               ;----- Read the old record
147
148   0240                        CHANGE_RECORD:
149   0240  3C 41                       CMP    AL,'A'                        ; Test character to be in
150   0242  7C E9                       JL     KEYBOARD_ERROR                ;  range 'A' - 'Z'
151   0244  3C 5A                       CMP    AL,'Z'
152   0246  77 E5                       JA     KEYBOARD_ERROR
153   0248  2A E4                       SUB    AH,AH                         ; Convert the character to
154   024A  2C 41                       SUB    AL,'A'                        ;  a record index
155   024C  A3 007D R                   MOV    WORD PTR FCB_RANDOM_RECORD,AX ; Store in FCB
156   024F  BA 005C R                   MOV    DX,OFFSET FCB
157   0252  B4 21                       MOV    AH,21H                        ; Random read of the record
158   0254  CD 21                       INT    21H
159   0256  0A C0                       OR     AL,AL                         ; Test for error
160   0258  74 06                       JE     RANDOM_READ_OK
161   025A  BA 014F R                   MOV    DX,OFFSET BAD_READ_MSG        ; Read failed error msg
162   025D  E9 01CD R                   JMP    ERROR_EXIT
163
164                               ;----- Display the old record
165
166   0260                        RANDOM_READ_OK:
167   0260  C6 06 00B0 R 0A             MOV    DISK_TRANSFER_ADDRESS+32,10   ; Put in carriage return,
168   0265  C6 06 00B1 R 0D             MOV    DISK_TRANSFER_ADDRESS+33,13   ;  line feed and string
169   026A  C6 06 00B2 R 24             MOV    DISK_TRANSFER_ADDRESS+34,'$'  ;  terminator
170   026F  B4 09                       MOV    AH,9
171   0271  BA 0090 R                   MOV    DX,OFFSET DISK_TRANSFER_ADDRESS
172   0274  CD 21                       INT    21H                           ; Display the record
173
174                               ;----- Update the record
175
176   0276  A0 0106 R                   MOV    AL,KEYBOARD_BUFFER+3          ; Get the replacement
177   0279  B9 001F                     MOV    CX,RECORD_SIZE-1              ;  character
178   027C  BF 0091 R                   MOV    DI,OFFSET DISK_TRANSFER_ADDRESS+1
179   027F  F3/ AA                      REP    STOSB                         ; Fill last 31 positions
180   0281  B4 22                       MOV    AH,22H
181   0283  BA 005C R                   MOV    DX,OFFSET FCB
182   0286  CD 21                       INT    21H                           ; Random write new record
183   0288  0A C0                       OR     AL,AL                         ; Test for error
184   028A  74 06                       JZ     RANDOM_WRITE_OK
185   028C  BA 0134 R                   MOV    DX,OFFSET BAD_WRITE_MSG       ; Write failed error message
186   028F  E9 01CD R                   JMP    ERROR_EXIT
187   0292                        RANDOM_WRITE_OK:
188   0292  EB 88                       JMP    KEYBOARD_LOOP                 ; Go back for another record
189
190                               ;----- End the program
191
192   0294                        PROGRAM_EXIT:
193   0294  B4 10                       MOV    AH,10H                        ; Close the file
194   0296  BA 005C R                   MOV    DX,OFFSET FCB
195   0299  CD 21                       INT    21H
196   029B  0A C0                       OR     AL,AL                         ; Test for error
197   029D  74 06                       JZ     CLOSE_OK
198   029F  BA 0167 R                   MOV    DX,OFFSET BAD_CLOSE_MSG       ; Close failed error msg
199   02A2  E9 01CD R                   JMP    ERROR_EXIT
200   02A5                        CLOSE_OK:
201   02A5  CD 20                       INT    20H                           ; Exit to DOS
202   02A7                        CODE      ENDS
203                                         END
```

Abbildung 5.6 Beispiel für DOS-Funktionen

Das Programmbeispiel besteht aus zwei Abschnitten. Im ersten Teil erstellt es eine Datei. Die Datei besteht 26 Datensätze von je 32 Bytes. Jeder Datensatz enthält einen einzelnen Buchstaben des Alphabets. Datensatz 1 ist „AAAA...A", Datensatz 2 ist „BBBB...B" usw. Die Datei wird sequentiell erstellt.

Der zweite Teil des Beispiels behandelt die Datei als eine Direktzugriffsdatei. Als Antwort auf die Benutzereingabe über die Tastatur liest das Programm einen der 26 Datensätze und gibt ihn aus. Die Eingabe über die Tastatur editiert dabei auch den Datensatz. Das Programm ändert die hinteren 31 Zeichen des Datensatzes auf den Wert, der von der Tastatur eingegeben wird. Die Eingabe von „$" beendet das Programm.

Das Beispiel simuliert eine Abfragedatenbank. Der erste Teil erstellt die Datenbank. Die zweite Phase befaßt sich mit direkten Anfragen an die Datenbank und mit dem Editieren der Daten. Jedes wirkliche Datenbankprogramm wäre sehr viel komplexer als dieses Beispiel hier, aber das Beispiel zeigt die hauptsächlichen Dateifunktionen.

Das Programm der Abbildung 5.6 ist eine .COM-Datei. Im nächsten Abschnitt wollen wir die Unterschiede zwischen dieser Datei und der .EXE-Datei besprechen. Der Gebrauch einer .COM-Datei erlaubt es uns hier, INT 20H als Programmausgang zu verwenden. Für eine .COM-Datei muß das Programm mit Offset 100H im Segment beginnen. Die ersten 100H Bytes des Programmsegmentes werden Programmsegment-Präfix (PSP) genannt und enthalten einige spezielle Felder zur Verwendung durch das Assemblerprogramm.

Der FCB befindet sich bei Offset 05CH im Segment. Das Programm benutzt diesen anscheinend beliebigen Ort aus gutem Grunde. Der DOS-Kommandointerpreter füllt diesen FCB auf. Wenn der Benutzer das Programm durch Eingeben seines Namens aufruft, durchsucht DOS den Rest der Kommandozeile auf Dateinamen. Es setzt den ersten Dateinamen, der dem Kommando folgt, in den FCB bei 05CH. Erscheint ein zweiter Name in der Kommandozeile, wird er in einen FCB mit Offset 06CH gebracht. Da dieses Beispiel nur mit einer Datei arbeitet, wird nur der FCB bei 05CH benutzt. Der Befehl für den Start unseres Programmes lautet:

A>FIG5-6 TEST.FIL

FIG5-6 ist der Name des Programms. Tatsächlich erscheint es auf der Diskette als FIG5-6.COM. Die Datei TEST.FIL ist die Datei, die vom Programm erstellt und später modifiziert wird. Der Kommandointerpreter bringt den Dateinamen TEST.FIL in die richtige Position im FCB bei 05CH. Da der Dateiname zum Kommandoparameter gemacht wurde, kann unser Programm jede Datei erstellen und modifizieren. Hätten wir den Dateinamen in das Assemblerprogramm einbezogen, hätten wir nur mit einer einzigen Datei arbeiten können.

Bei Offset 05CH gibt das Programm die Struktur des FCB wieder. Die Bezeichnung FCB identifiziert den ersten Speicherplatz. Jedes der Felder im FCB hat seine eigene Bezeichnung und Größe, so daß das Programm sie individuell behandeln kann. Um zum Beispiel die Größe des Datensatzes einzustellen, modifiziert das Programm die Variable FCB_RECORD_SIZE.

Bei Offset 080H liegt ein weiteres spezielles Feld des PSP. Dieser 128 Bytes umfassende Bereich ist die voreingestellte Speicherstelle für die Disk Transfer Area (DTA). DOS benutzt die DTA als Pufferbereich für Datensätze. Immer wenn DOS einen Datensatz liest oder schreibt, benutzt es den DTA-Puffer. DOS initialisiert die DTA bei Offset 080H des Programmsegments. Das Programm kann dies ändern, indem es die Funktion 1AH von Interrupt 21H benutzt. Ist die Satzlänge größer als 128 Bytes, muß der voreingestellte Wert geändert werden. Das Beispiel ändert den DTA-Offset in Segment auf 90H. Dies ist notwendig, weil sich der FCB bei 05CH über die Speicherstelle 80H hinaus ausdehnt. Würde die voreingestellte DTA benutzt, würde die Datenübertragung das letzte Byte des FCB überschreiben. Da dieses Byte die Satznummer für den Direktzugriff enthält und unser Programm den Direktzugriff benutzt, ist dieses Vorgehen notwendig, um eine Kollision der Daten zu vermeiden.

Die erste Anweisung des Programms bei Offset 100H ist der Sprung zum eigentlichen Beginn des Programms. Dies mag unwirtschaftlich erscheinen, aber der Assembler erledigt das Assemblieren sehr viel besser, wenn alle Datenbereiche vor den Befehlen erscheinen, die sie ansprechen. Das Programm kann in der Tat Fehler enthalten, wenn es Daten anspricht, die erst später definiert werden. Um der Sicherheit willen erscheint der Datenbereich also am Anfang des Programms.

Der erste Programmabschnitt belegt die DTA. Unser Programm benutzt den Puffer bei Offset 90H. Da der Umfang des Datensatzes nur 32 Bytes beträgt, ist vor Beginn des Programms viel Platz.

Als nächstes benutzen wir eine Funktion von Interrupt 21H, um nach einer Datei mit dem gewünschten Namen im FCB zu suchen. Merken Sie sich, daß DS:DX auf den FCB hinweist, wie das für jede Dateioperation der Fall sein wird. Wenn DOS eine Datei mit einem übereinstimmenden Namen findet, steigt das Programm mit einer Fehlermeldung aus anstatt die bereits bestehende Datei zu löschen. Dieses Beispiel funktioniert also nur mit einer neuen Datei und kann nicht auf eine bereits bestehende angewandt werden. Diese Überprüfung schließt aus, daß das Programm eine bestehende Datei löschen kann. Aber wenn unser Beispiel ein echtes Programm wäre, würden wir vermutlich mit beiden, neuen und bestehenden Dateien arbeiten.

Bei NO_FILE erstellt das Programm die Datei. Da der Kommandoprozessor den FCB bereits aufgebaut hat, muß unser Programm ihn nicht vor der Dateikreation neu belegen. Wenn die Operation aus irgendeinem Grund fehlschlägt, weil z.B. kein Platz mehr im Inhaltsverzeichnis oder auf der Diskette vorhanden ist, steigt das Programm mit einer eigenen Fehlermeldung aus.

Immer wenn DOS auf eine Datei zugreifen soll, muß das Programm die Datei zuerst eröffnet haben. Das Eröffnen der Datei stellt eine Verbindung zwischen dem Betriebssystem und dem Benutzerprogramm her. Während der Eröffnung durchsucht DOS das Inhaltsverzeichnis der Diskette, findet die Datei (oder vielleicht auch nicht, dann tritt ein Fehlerzustand auf) und belegt das Feld des FCB, das sich auf den Umfang der Datei bezieht. Wenn DOS die Datei einmal eröffnet hat, braucht es das Inhaltsverzeichnis nicht bei jedem Zugriff auf die Datei zu durchsuchen. DOS behält die Verbindungsinformation zur Datei im FCB, bis es die Datei „abschließt". Aus den

Ausdrücken „Öffnen" und „Schließen" können Sie erneut den Einfluß des bisher üblichen Umgangs mit Akten erkennen. Die Akte muß geöffnet werden, bevor die Unterlagen gelesen werden können. Die Akte wird geschlossen, bevor man sie wieder in den Aktenschrank zurückstellt.

Unser Beispiel eröffnet die Datei dadurch, daß es sie erstellt. Wenn die Datei bereits bestanden hätte, hätte die Eröffnungsfunktion (AH=0FH) die Verbindung hergestellt. Nachdem die Datei erfolgreich eröffnet ist, verändert das Programm einige der Felder im FCB. Im besonderen muß es die Größe des Datensatzes auf 32 Bytes einstellen, da das DOS eine Satzlänge von 128 Bytes voreingestellt hat.

Der Abschnitt CHARACTER_LOOP schreibt die 26 Datensätze in die Datei. Ein REP STOSB füllt den DTA-Zwischenspeicher mit dem Zeichen für diesen Datensatz. Eine sequentielle Schreibfunktion (AH=0AH) schreibt die individuellen Datensätze. Das Programm prüft dabei auch auf Fehler.

Bei KEYBOARD_LOOP wechselt der Programmodus von Erstellung auf Abfrage/ Aktualisierung. Das Beispiel benutzt dazu die von DOS ermöglichte gepufferte Tastatureingabe. Diese Funktion gestattet es dem Benutzer, einen 2-Byte String einzugeben und diesen, wenn notwendig, zu editieren. Der 2-Byte String muß mit einem Return-Zeichen abgeschlossen sein. Das Programm prüft die Eingabe auf Gültigkeit und weist sie mit einer Fehlermeldung zurück, wenn sie den Eingabeerfordernissen nicht entspricht.

Gibt der Benutzer „$" ein, springt das Programm zum Ende. Bei einem Zeichen zwischen „A" und „Z" liest das Programm den Datensatz entsprechend dem Zeichen. Der Datensatz erscheint auf dem Bildschirm und unser Programm füllt die restlichen 31 Bytes des Puffers mit dem zweiten Zeichen, welches über die Tastatur eingegeben wurde. Danach wird der modifizierte Datensatz direkt zurückgeschrieben.

Der letzte Abschnitt des Programms schließt die Datei. Genau wie die Dateieröffnung die Verbindung zwischen DOS und dem Benutzerprogramm hergestellt hat, trennt die Close-Funktion diese Verbindung. Ein wichtiger Grund für den Abschluß einer Datei ist, sicherzugehen, daß DOS alle modifizierten Datensätze auf die Diskette geschrieben hat. Während einer normalen Programmausführung kann DOS die letzten Datensätze in einem Puffer belassen. Dies beschleunigt die Ausführung, weil DOS dann nicht für jeden Datensatz auf die Diskette zu schreiben braucht. Die Close-Funktion schreibt den Inhalt dieses Puffers dann auf die Diskette.

Das Beispiel der Abbildung 5.6 zeigt die grundlegenden Komponenten des Dateizugriffs unter Benutzung von DOS. Das Programm tut nichts Nützliches, doch es würde zu vieler weiterer Instruktionen bedürfen, um es vernünftig auszubauen. Solche Instruktionen würden wenig bringen, was den Gebrauch von DOS-Funktionen anbelangt. Eine wichtige Sache, die man bei diesem Beispiel beachten sollte, ist jedoch die Notwendigkeit des Prüfens auf Fehler nach jeder DOS-Operation. Während DOS Hardwarefehler mit der Unterbrechung 24H prüft, muß das Anwenderprogramm Fehler wie gleiche Dateinamen oder Platzmangel auf der Diskette selbst beheben. Die Fehlerbehandlung ist im Beispiel sehr einfach — Ausgabe einer Fehlermeldung und Programmbeendigung. In echten Programmen ist die Fehlerbehandlung sehr viel komplexer und sehr viel wichtiger. Der Verlust wertvoller Daten muß nämlich unter allen Umständen vermieden werden.

Und wenn Sie schließlich das Beispielprogramm ausführen, werden Sie über dessen Anforderungen an den Benutzers vermutlich nicht sehr glücklich sein. Es gibt keine Eingabeanforderung für Benutzereingaben, die Fehlermeldungen sind beinahe unverständlich, und einige der Meldungen werden schließlich über die vorhergehenden geschrieben und löschen sie damit aus. Das Programm erfordert also weitere Bearbeitung, bevor es von jemandem benutzt werden könnte, der es nicht selbst geschrieben oder aber sehr sorgfältig studiert hat.

.COM- und .EXE-Dateien

Das vorangegangene Beispiel war eine .COM-Datei. Aber die normale Ausgabe des Assemblier-/Linkprozesses ist eine .EXE-Datei. Warum plagen wir uns also mit .COM, wenn .EXE einfacher ist?

Jeder dieser Dateitypen hat seine Vorteile. Wir müssen die zwischen ihnen bestehenden Unterschiede verstehen, um richtig zu entscheiden, welche Form in welchem Fall zu wählen ist.

Der Hauptunterschied zwischen .COM- und .EXE-Dateien ist das Format der Datei auf der Diskette. Beide Dateitypen enthalten Maschinencode-Programme. Die .COM-Datei kann sofort benutzt werden. DOS kann sie direkt von der Diskette in den Speicher laden. DOS übergibt die Steuerung dann an die Stelle mit Offset 100H in dem Segment, das für das Programm reserviert worden ist. Die .EXE-Datei ist dagegen nicht sofort ausführbar. Der Maschinencode in der Datei hat ein Vorsatz- oder „Header"-Feld. Dieser Header enthält vom Linker erstellte Informationen. Das Bemerkenswerte daran ist die Verschiebeinformation. Während eine .COM-Datei nur über eine Änderung des Codesegments verschiebbar ist, kann eine .EXE-Datei über viele verschiedene Segmente gespeichert werden. Dies begrenzt die .COM-Datei auf eine maximale Größe von 64K Bytes, sofern das Programm nicht auch die anderen Segmente selbst lädt. Eine .EXE-Datei kann zahlreiche Segmente enthalten, die bei Bedarf dynamisch zugeteilt bzw. benützt werden.

Was ist Verschiebbarkeit? Wenn ein Programm assembliert ist, hat es seinen festen Platz im Speicher. Wir haben gesehen, daß der Assembler automatisch jedes Segment bei Offset 0 beginnt. In den Assemblerlisten findet man neben einigen Adressen das Zeichen „R". Dies zeigt, daß die Adresse verschiebbar ist. Wenn das Programm verschoben wird, so daß es an einem anderen Punkt als mit Offset 0 beginnt, muß diese Adresse modifiziert werden. Normalerweise behandelt der Linker diese Verschiebung. Aber einige Adressen können nicht modifiziert werden, bevor das Programm geladen ist. Eine .EXE-Datei enthält nun die nötigen Daten, um diese Stellen zu identifizieren.

Eine .COM-Datei ist nicht verschiebbar. Sie verfügt über keine Verschiebeinformation. Stattdessen ist ein .COM-Programm über das Codesegment verschiebbar. Das bedeutet, daß das Programm immer mit dem gleichen Offset geladen wird, während das Codesegment modifiziert werden kann. Alle Offsets im Programm bleiben gleich. Dies erfordert außerdem, daß der Programmierer sichergeht, daß jede Segmentregisterarithmetik (z. B. das Einsetzen eines direkten Wertes in ein Segmentre-

gister) sich immer auf das aktuelle Code-Segmentregister bezieht. Um z. B. das DS-Register auf den aktuellen Wert des Codesegments einzustellen, lautet die richtige Befehlsfolge

```
PUSH    CS
POP     DS
```

Man ist manchmal versucht, den folgenden Code für den gleichen Vorgang zu schreiben, wobei wir annehmen, daß das Codesegment wie in Abbildung 5.6 „CODE" genannt wird:

```
MOV     AX,CODE
MOV     DS,AX
```

Dieses Beispiel arbeitet in einem .COM-Programm nicht korrekt. Der Segmentwert für das Segment CODE ist zum Zeitpunkt des Assemblierens und Linkens nicht bekannt. Er ist nur bekannt, wenn das Programm geladen wird. Da die .COM-Datei dem Lader nicht sagen kann, wo all diese Segment-Bezugsadressen erscheinen (dazu wäre die Verschiebeinformation nötig), wird das Programm unrichtig ausgeführt.

Das Setzen der Segmentregister und Stackadressen erfolgt unterschiedlich für die zwei Dateitypen. Für eine .COM-Datei setzt DOS das CS-, DS-, ES- und SS-Register auf das Segment, in welches es das Programm lädt. Es setzt das SP-Register als Zeiger auf den letzten verfügbaren Speicherplatz im Segment. Auf diese Weise liegt das Programm im vorderen Teil des Segmentes und der Stack im hinteren.

Der .EXE-Dateiheader spezifiziert die Werte für das CS-, IP-, SS- und SP-Register. DOS setzt das DS- und ES-Register auf das Segment, in welches es das Programm lädt. Das CS-Register deutet auf das Segment, das den Ort des Programmbeginns enthält. Während ein .COM-Programm bei Offset 100H des Codesegments beginnen muß, kann ein .EXE-Programm einen anderen Startort spezifizieren. Der Assemblerbefehl END kann dafür einen Adresswert enthalten, wie nachfolgend gezeigt:

```
END     START_LOCATION
```

Dies sagt dem Assembler/Linker, die Steuerung an die Stelle START_LOCATION zu übergeben, wenn das Programm geladen ist.

Beide Dateitypen benutzen das Programmsegment-Präfix (PSP). Dies sind die ersten 100H Bytes des Segments, in welches das Programm geladen ist. Dieser Bereich enthält die speziellen Felder, die wir in Abbildung 5.6 gesehen haben. Für eine .EXE-Datei zeigen DS- und ES-Register auf diesen Bereich, während CS und SS-Register durch den Assemblier-/Linkvorgang gesetzt werden. Bei der .COM-Datei zeigen alle Register auf das PSP. Dies gestattet beiden Programmtypen unmittelbaren Zugriff auf die Daten im PSP.

Der Vorteil der .COM-Datei liegt darin, daß das CS-Register das PSP bestimmt. In einer .EXE-Datei ist dies nicht der Fall. Die DOS-Programmausgänge über Interrupt 20H und 27H erfordern aber, daß das CS-Register auf PSP zeigt, wenn die Unterbrechung ausgeführt wird. Mit einer .EXE-Datei ist dies schwierig. Glücklicherweise

erlaubt es die nachstehende Instruktionsfolge einem .EXE-Programm, zu DOS zurückzukehren.

```
PROGRAM    PROC    FAR
           PUSH    DS          ; sichere Segment von PSP
           MOV     AX,0
           PUSH    AX          ; sichere Offset 0 in Stack
           ...
           RET
PROGRAM    ENDP
```

Bei Offset 0 des PSP finden wir den Befehl INT 20H. PUSH DS und 0 stellen eine Rückkehradresse für RETF auf Offset 0 des PSP her. Wenn das Programm die Rückkehr ausführt, kommt es zu INT 20H. Jetzt zeigt das CS-Register aber auf PSP, und INT 20H übergibt die Steuerung wieder an DOS.

Es gibt keinen vergleichbaren Weg, einen Interrupt 27H - Rückkehr zu DOS, wobei das Programm resident bleibt - auszuführen. Obgleich es Möglichkeiten gibt, das CS-Register korrekt einzustellen bevor INT 27H ausgeführt wird, ist es in der Regel einfacher, das Programm als .COM-Datei zu aufzubauen.

Schließlich braucht eine .COM-Datei auch noch weniger Platz auf der Diskette als eine .EXE-Datei mit demselben Programm. Da die .COM-Datei keinen Programmheader hat, ist dafür auch kein Platz erforderlich. Wenn wir im nächsten Abschnitt das DEBUG-Programm besprechen, schließt das auch eine Methode mit ein, mit der Sie eine .EXE-Datei in eine .COM-Datei umwandeln können.

Das Erstellen eines Assemblerprogramms

Es gibt mehrere Schritte auf dem Wege von der Idee zu einem laufenden Programm. Dieser Abschnitt behandelt die Schritte, die sich auf das Erstellen eines Assemblerprogramms für den IBM PC beziehen. Wir behandeln dabei den Zeileneditor, den Assembler, den Linker und den Debugger. Mit dem Editor erstellen wir das Quellprogramm in Assemblersprache. Der Assembler wandelt dieses Quellprogramm in einen Maschinencode um, der fast schon der eigentlichen Maschinensprache gleicht. Der Linker vollendet diesen Maschinencode zu einer ausführbaren .EXE-Datei. Das DEBUG-Programm kann schließlich noch dazu beitragen, Fehler im Programm zu finden.

DOS-Zeileneditor

Der Zeileneditor erstellt Textdateien. Der Inhalt einer Textdatei wird in ASCII dargestellt. Der Editor läßt Sie den gewünschten Text in die Datei eintippen. Wenn Sie den Text später modifizieren müssen, benutzen Sie wieder den Editor.

Der Zeileneditor EDLIN ist Teil des IBM PC Disk Operating Systems. Die Datei EDLIN.COM ist ein transientes Kommando — das heißt, es wird nur dann in den

Speicher geladen, wenn es verlangt wird. Um mit dem Editor zu arbeiten, lautet der Befehl

A>EDLIN FILE.ASM

wobei FILE.ASM die zu editierende Textdatei ist. FILE.ASM kann schon vorhanden sein, wie z. B., wenn Sie ein bestehendes Programm modifizieren. Ist die Datei FILE.ASM noch nicht vorhanden, wird sie durch EDLIN erstellt. Wenn das Editieren vollständig ausgeführt ist, wird das Ergebnis in die Datei FILE.ASM übertragen. Sofern EDLIN die Datei FILE.ASM für diesen Editiervorgang nicht neu erstellt hat, gibt er der alten Version von FILE.ASM zur Sicherung den Namen FILE.BAK. Durch das Anlegen einer Sicherung können Sie auch nach größeren Fehlern beim Editieren der Datei wieder weitermachen, indem Sie die Sicherungsdatei benutzen. Sie löschen die Datei mit Editionsfehlern und benennen die .BAK-Datei wieder in .ASM-Datei um. Dies ist also eine Art Sicherheitsnetz beim Editieren.

EDLIN ist ein zeilenorientierter Editor. Er behandelt jede Textzeile getrennt, im Gegensatz zu einem bildschirmorientierten Editor, der einen Abschnitt der aufbereiteten Datei auf dem Bildschirm darstellt. Bei einem bildschirmorientierten Editor, wie dem IBM Personal Editor, verschieben Sie den Cursor an irgendeine Stelle auf dem Bildschirm und modifizieren dort den Text. Mit EDLIN editieren Sie immer nur eine einzelne Zeile der Textdatei. Wenn Sie ernsthaftes Assemblerprogrammieren beabsichtigen, sollten Sie einen bildschirmorientierten Texteditor benutzen. IBM bietet einen sehr guten Editor unter der Bezeichnung „Personal Editor" an. Die Beispiele in diesem Buch beziehen sich auf die Benutzung dieses Editors. Wir besprechen EDLIN nur, weil er zusammen mit DOS verkauft wird. Nachdem EDLIN aber nicht gerade die beste Wahl ist, haben wir für EDLIN lediglich eine kurze Erwähnung der Kommandos und ein Beispiel vorgesehen. Für eine vollständige Erklärung der EDLIN-Kommandos sollten Sie das DOS Manual zu Rate ziehen.

Abbildung 5.7 faßt die EDLIN-Kommandos zusammen. Beim Besprechen des Beispiels werden wir auf diese Kommandos Bezug nehmen. Abbildung 5.8 zeigt zwei Beispiele für Editiervorgänge unter Einsatz von EDLIN.

Aktion	Syntax
Zeile(n) anfügen	[n] A
Zeile(n) löschen	[Zeile] [,Zeile] D
Zeile ausgeben	[Zeile]
Beenden	E
Zeile einfügen	[Zeile] I
Zeile(n) auflisten	[Zeile] [,Zeile] L
Editor verlassen	Q
Text ersetzen	[Zeile] [,Zeile) [?] RString [<F6>String]
Zeile(n)) schreiben	[n] W

Abbildung 5.7 EDLIN-Kommandos
(mit freundlicher Genehmigung der IBM; Copyright IBM 1981)

```
A>EDLIN FIG5-8.ASM
New file
*I
        1:*        PAGE       ,132
        2:*CODE    SEGMENT
        3:*        ASSUME     CS:CODE
        4:*
        5:*START:  PROC       NEAR
        6:*        MOV        AX,CODE
        7:*        MOV        DS,AX
        8:*        RET
        9:*START   ENDP
       10:*END
       11:*^C
*E
A>
A>EDLIN FIG5-8.ASM
End of input file
*L
        1:*        PAGE       ,132
        2: CODE    SEGMENT
        3:         ASSUME     CS:CODE
        4:
        5: START:  PROC       NEAR
        6:         MOV        AX,CODE
        7:         MOV        DS,AX
        8:         RET
        9:
       10: START   ENDP
       11: END
*3
        3:*        ASSUME     CS:CODE
        3:*        ASSUME     CS:CODE,DS:CODE
*5
        5:*START:             PROC       NEAR
        5:*START   PROC       NEAR
*6
        6:*        MOV        AX,CODE
        6:*        MOV        AX,CS
*9I
        9:*
       10:*^C
*L
        1:         PAGE,132
        2: CODE    SEGMENT
        3:         ASSUME     CS:CODE,DS:CODE
        4:
        5: START   PROC       NEAR
        6:         MOV        AX,CS
```

```
          7:          MOV       DS,AX
          8:          RET
          9:
          10:*START   ENDP
          11: END
*E
A>
```

Abbildung 5.8 EDLIN-Beispiel

Im ersten Teil erstellen wir eine Datei mit dem Namen FIG5-8.ASM. „New file" zeigt an, daß es diese Datei bisher noch nicht gegeben hat. Das Zeichen „*" ist die Eingabeanforderung für EDLIN. Immer wenn dieses Zeichen erscheint, können Sie irgendein Kommando aus der Abbildung 5.7 eingeben. Das „I"-Kommando erlaubt es Ihnen, Text in die Datei einzufügen. EDLIN stellt die Zeilennummern für jede Zeile dar, wenn Sie den Text eingeben. Control-C (^C auf der Liste) beendet den eingefügten Teil. Das „E" beendet den Editiervorgang, sichert die Datei und gibt die Steuerung an DOS zurück.

Im zweiten Teil von Abbildung 5.8 editieren wir die im ersten Teil erstellte Datei. Sie rufen den Editor mit derselben Kommandozeile auf, aber der Ausdruck „End of input file" (zu deutsch: Ende der Eingabedatei) gibt an, daß die Datei bereits vorhanden ist. Das Kommando „L" listet die Datei in ihrem aktuellen Stand auf. Das Beispiel benutzt einzeilige Editierkommandos, um mehrere der Zeilen zu modifizieren. Das Eingeben einer „3" zeigt die dritte Zeile und hält sie für weiteres Editieren frei. DOS hat einen eingebauten Satz von Zeilen-Editierkommandos, die in Abbildung 5.9 dargestellt sind. Sie können diese Editionsoperationen in EDLIN benutzen, oder jedesmal, wenn DOS mit der Eingabe über die Tastatur arbeitet. Diese Befehle sind also insbesondere für die Eingabe an den Kommandoprozessor geeignet.

Gehen wir zurück zum Beispiel in Abbildung 5.8. Nach der Darstellung der Zeile mit ihrer Zeilennummer können Sie die Zeile mit DOS-Kommandos editieren. Die F3-Taste zeigt die Zeile erneut und ermöglicht es Ihnen, die Änderungen am Zeilenende einzugeben. Das Editieren der Zeile 5 zeigt den Gebrauch der DEL-Taste, um den ":" aus dem Label zu entfernen. In Zeile 6 benutzen wir die → Taste (Cursor rechts), um den Cursor hinter das Komma zu setzen. Dann wird der neue Wert „CS" eingetippt, der „CODE" ersetzen soll. Das Beispiel macht eine Einfügung vor Zeile 9, - eine Leerzeile. Dann wird die Datei wieder so aufgelistet, daß Sie die Änderungen durch den Editiervorgang erkennen können. Das Editierkommando „E" gibt die Steuerung an DOS zurück.

Sie können mit EDLIN und den DOS-Editierkommandos auch andere Vorgänge ausführen. Sie lernen den Umgang mit ihnen am besten, wenn Sie das DOS Manual durchsehen und dann einige davon einfach ausprobieren. Für ernsthafte Programmierarbeit sollten Sie jedoch den Erwerb eines bildschirmorientierten Editors in Erwägung ziehen.

DEL	**Löschen nächstes Zeichen in aktueller Zeile**
ESC	**Originalzustand der aktuellen Zeile wiederherstellen**
F1 oder ->	**Kopieren ein Zeichen aus der aktuellen Zeile**
F2	**Kopieren alle Zeichen bis zu einem bestimmten**
F3	**Ausgeben aller restlichen Zeichen auf den Bildschirm**
F4	**Löschen aller Zeichen bis zu einem bestimmten**
F5	**weiteres Editieren in der aktuellen Zeile**
INS	**Zeichen in aktuelle Zeile einfügen (bis INS wieder gedrückt wird)**

Abbildung 5.9 DOS-Editiertasten
(mit freundlicher Genehmigung der IBM; Copyright IBM 1981)

Assembler und Makroassembler

Nun, da Sie die Quelldatei erstellt haben, ist es an der Zeit, den Assembler zu benutzen. Es gibt zwei Assemblerversionen. Die vollständige Version ist der Makroassembler oder MASM.EXE auf der Assemblerdiskette. Darüber hinaus gibt es eine kleinere Version des Assemblers ohne Makroverarbeitung, genannt Assembler oder ASM.EXE. MASM benötigt 96K Bytes Speicher, um effektiv zu laufen, während ASM mit nur 64K auskommt. Dieser Speicherbedarf hat nichts mit dem Umfang des Quellprogrammes zu tun. Er bezieht sich lediglich auf die erforderliche Speichergröße für das Assemblieren des Programmes, nicht aber auf die Durchführung desselben. Ein Benutzerprogramm, das für seine Ausführung möglicherweise nur 4K Bytes benötigt, braucht eine Maschine mit mindestens 64K, um das Programm entwickeln zu können.

Die Eingabe für den Assembler ist die Quelldatei, die durch EDLIN oder einen ähnlichen Editor erstellt wurde. Die Quelldatei ist eine ASCII-Textdatei. Der Assembler selbst produziert bis zu drei Ausgabedateien. Die Objektdatei enthält die maschinensprachliche Version des Programms. Sie ist allerdings noch nicht ganz fertig für die Ausführung, aber sie kommt dem endgültigen Maschinencode schon sehr nahe. Die Listendatei ist eine ASCII-Textdatei, die sowohl die Quellinformation als auch die durch den Assembler erstellte, neue Information enthält. Die Beispiele in diesem Buch sind aus Assembler-Listendateien. Schließlich kann der Assembler auch eine Querverweisdatei herstellen. Diese Datei, welche weder eine Objekt- noch eine Textdatei ist, enthält Information über den Gebrauch von Symbolen und Kennzeichen im Assemblerprogramm. Gleich der Objektdatei erfordert auch die Querverweisdatei eine weitere Bearbeitung, bevor Sie sie benutzen können.

Der Assembler wird mit einem DOS-Kommando gestartet. Entweder startet

```
A>ASM
```

oder

```
A>MASM
```

den Assembler. ASM ruft den kleinen Assembler auf, MASM ruft den Makroassembler auf. Wenn der Assembler mit der Ausführung beginnt, stellt er Fragen, um zu

bestimmen, welche Dateien für die Assemblierung benutzt werden sollen. Abbildung 5.10 zeigt die Reihenfolge der Kommandos, die den Assembler starten.

Nach dem Kommando ASM lädt DOS den Assembler in den Speicher. Der Assembler gibt die Copyright-Meldung aus und beginnt mit seinen Fragen an den Benutzer. Wenn Sie ein System mit einem einzelnen Diskettenlaufwerk haben, können Sie jetzt die Assemblerdiskette entfernen und die Datendiskette einlegen. Der Assembler stellt die Eingabeanforderung für die Datei, die assembliert werden soll. Sie müssen lediglich den Dateinamen eingeben, nicht die .ASM-Erweiterung. Der Assembler fragt auch nach den Namen für die Ausgabedateien. Der Assembler gibt der Objektdatei denselben Namen wie der Quelldatei, aber mit einer .OBJ-Erweiterung, wenn Sie nicht einen anderen Namen dafür wählen. Die „B:"-Antwort in diesem Beispiel sagt dem Assembler, die Objektdatei im Laufwerk B: abzulegen. Ähnliche Antworten auf die Eingabeanforderung für Liste und Querverweis sagen dem Assembler, diese Dateien im Laufwerk B: abzulegen. Nach dem Assemblieren zeigt das Inhaltsverzeichnis die Dateien im Laufwerk B:, die durch diesen Vorgang erstellt wurden.

```
A>ASM
The IBM Personal Computer Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Source filename [.ASM]: B:FIG5-10
Object filename  [FIG5-10.OBJ] B:
Source listing   [NUL.LST] B:
Cross reference [NUL.CRF] B:

Warning Severe
Errors   Errors
0        0

A>DIR B:FIG5-10.*
FIG5-10  ASM     44    1-01-83  12:00a
FIG5-10  LST    426    1-01-83  12:00a
FIG5-10  OBJ     40    1-01-83  12:00a
FIG5-10  CRF     19    1-01-83  12:00a

A>
A>B:
B>A:ASM FIG5-10,,,;
The IBM Personal Computer Assembler
Version 1.00 (C) Copyright IBM Corp 1981

Warning   Severe
Errors    Errors
0         0

B>
```

Abbildung 5.10 Assemblerlauf

Die Eingabeanforderungen haben alle voreingestellte Werte, die in Klammern angegeben sind. Wenn Sie mit einem „Return" auf eine dieser Eingabeanforderungen antworten, benutzt der Assembler den voreingestellten Wert. Der Default für Listen- und Querverweisdatei ist NUL. Die NUL-Datei ist eine spezielle Datei im DOS. Alles, was auf diese NUL-Datei geschrieben wird, verschwindet und kann nicht mehr angesprochen werden. Die NUL-Datei ist eine WOF-Datei (Write-only file).

Wenn der Assembler während des Assemblierens irgendwelche Fehler entdeckt, schreibt er diese auf die Listendatei. Er zeigt sie außerdem auf dem Bildschirm an. Dies ermöglicht es Ihnen, auf entdeckte Fehler direkt zu reagieren. Sie müssen also die Listendatei nicht nach Fehlern durchsuchen. Wenn Sie mit dem kleinen Assembler ASM arbeiten, zeigt dieser die Fehler lediglich mit einer Fehlernummer an. Der Makroassembler MASM zeigt dagegen Fehlernummer und Fehlertext. Die kleinere Kapazität des Assemblers läßt nämlich keinen Raum für den Fehlermeldungstext.

Der zweite Teil der Abbildung 5.10 zeigt eine einfachere Methode, den Makroassembler aufzurufen. Diese Methode ist günstig, wenn das von Ihnen benutzte System zwei Diskettenlaufwerke besitzt. Setzen Sie die Assemblerdiskette in das Laufwerk A:, die Datendiskette, die auch die Quelldatei enthält, in das Laufwerk B:. Setzen Sie das Laufwerk B: als das voreingestellte Laufwerk. Rufen Sie den Assembler mit dem Kommando A:ASM auf. Der Rest des Kommandos, FIG5-10,,,; gibt dem Assembler alle Information, die mit den Eingabeanforderungen für das vorhergehende Beispiel abgefragt wurde. FIG5-10 nennt die zu assemblierende Datei, während die nachfolgenden Kommas dem Assembler angeben, die Objekt-, Listen- und Querverweisdateien den normalen Kennzeichnungsrichtlinien entsprechend zu erstellen. Die Ergebnisse dieses Vorgehens sind mit denen des ersten Beispiels identisch.

Der Weg, auf dem Sie Dateinamen an den Assembler geben können, hat viele Variationen. Die beiden Beispiele hier stellen zwei Extreme dar. Das erste Beispiel spezifiziert jeden Dateinamen als Antwort auf eine Eingabeanforderung. Im zweiten war überhaupt keine Eingabeanforderung notwendig. Das Makroassembler-Handbuch beschreibt die mit dem ASM- (oder dem MASM-) Kommando möglichen verschiedenen Variationen sehr viel detaillierter.

Sobald das Assemblieren abgeschlossen ist, sind die Ausgabedateien verfügbar. Die Objektdatei ist die Eingabe für den nächsten Schritt in der Erstellung eines ladefähigen Programms. Dies ist der LINK-Schritt, der im nächsten Abschnitt behandelt wird.

Die Listendatei ist eine Kombination aus Quelldatei und einer lesefähigen Version der Maschinensprache. Sie können diese Datei auf dem Bildschirm darstellen mit dem DOS TYPE-Kommando, wie

```
A>TYPE B:FIG5-11.LST
```

Das TYPE-Kommando nimmt den Inhalt der Datei und sendet ihn zum Bildschirm. Gleichzeitig können Sie die Datei drucken, indem Sie Control-PRTSC drücken, bevor Sie das TYPE-Kommando geben. Control-PRTSC weist DOS an, alle Daten sowohl an den Bildschirm als auch an den Drucker zu übermitteln. Als Ergebnis

erscheint die Auflistung auf dem Bildschirm und auf dem Drucker. Sie sollten dabei angeben, daß die Druckausgabe 132 Schreibstellen breit ausgelegt sein soll. Sie tun dies mit dem Assembler PAGE-Kommando, welches Sie in fast jedem Programmbeispiel finden können. Das Kommando

PAGE ,132

sagt dem Assembler, die Listendatei 132 Zeichen breit zu machen. Vor dem Drukken müssen Sie außerdem die Zeilenbreite für den Drucker einstellen. Sie können dies mit dem DOS MODE-Kommando.

A>MODE LPT1:132

Das Kommando stellt den IBM-Drucker auf 132 Zeichen/Zeile ein. So wird die Datei ohne den auf dem Bildschirm sichtbaren Zeilenüberlauf ausgedruckt.

Symboltabelle

Die Listendatei enthält zusätzliche Information, die wir bis jetzt noch nicht erwähnt haben. Nach dem Auflisten des Programmes erscheint nämlich die Symboltabelle. Ein Beispiel wird in Abbildung 5.11 gezeigt. Es ist die Symboltabelle für das Pro-

```
The IBM Personal Computer Assembler 01-01-83         PAGE     Symbols-1
Figure 5.6  DOS Function Example

Segments and groups:

                N a m e                     Size    align   combine class

CODE . . . . . . . . . . . . . .            02A7    PARA    NONE

Symbols:

                N a m e                     Type    Value   Attr

BAD_CLOSE_MSG. . . . . . . . . .            L BYTE  0167    CODE
BAD_OPEN_MSG . . . . . . . . . .            L BYTE  011E    CODE
BAD_READ_MSG . . . . . . . . . .            L BYTE  014F    CODE
BAD_WRITE_MSG. . . . . . . . . .            L BYTE  0134    CODE
CHANGE_RECORD. . . . . . . . . .            L NEAR  0240    CODE
CHARACTER_LOOP . . . . . . . . .            L NEAR  01FC    CODE
CHAR_BAD_MSG . . . . . . . . . .            L BYTE  019E    CODE
CLOSE_OK . . . . . . . . . . . .            L NEAR  02A5    CODE
CREATE_OK. . . . . . . . . . . .            L NEAR  01E3    CODE
DISK_TRANSFER_ADDRESS. . . . . .            L BYTE  0090    CODE
ERROR_EXIT . . . . . . . . . . .            L NEAR  01CD    CODE
FCB. . . . . . . . . . . . . . .            L BYTE  005C    CODE
FCB_BLOCK. . . . . . . . . . . .            L WORD  0068    CODE
FCB_CURRENT_RECORD . . . . . . .            L BYTE  007C    CODE
FCB_DATE . . . . . . . . . . . .            L WORD  0070    CODE
FCB_DRIVE. . . . . . . . . . . .            L BYTE  005C    CODE
FCB_EXT. . . . . . . . . . . . .            L BYTE  0065    CODE    Length =0003
FCB_FILE_SIZE. . . . . . . . . .            L DWORD 006C    CODE
FCB_NAME . . . . . . . . . . . .            L BYTE  005D    CODE    Length =0008
FCB_RANDOM_RECORD. . . . . . . .            L DWORD 007D    CODE
FCB_RECORD_SIZE. . . . . . . . .            L WORD  006A    CODE
FCB_RESV . . . . . . . . . . . .            L BYTE  0072    CODE    Length =000A
FILE_ERROR_MSG . . . . . . . . .            L BYTE  0108    CODE
INPUT_BAD_MSG. . . . . . . . . .            L BYTE  017F    CODE
KEYBOARD_BUFFER. . . . . . . . .            L BYTE  0103    CODE
KEYBOARD_ERROR . . . . . . . . .            L NEAR  022D    CODE
KEYBOARD_LOOP. . . . . . . . . .            L NEAR  021C    CODE
KEY_INPUT_OK . . . . . . . . . .            L NEAR  0233    CODE
NO_FILE. . . . . . . . . . . . .            L NEAR  01D3    CODE
PROGRAM_EXIT . . . . . . . . . .            L NEAR  0294    CODE
PROGRAM_START. . . . . . . . . .            L NEAR  01B8    CODE
RANDOM_READ_OK . . . . . . . . .            L NEAR  0260    CODE
RANDOM_WRITE_OK. . . . . . . . .            L NEAR  0292    CODE
RECORD_SIZE. . . . . . . . . . .            Number  0020
WRITE_OK . . . . . . . . . . . .            L NEAR  0216    CODE

Warning Severe
Errors  Errors
0       0
```

Abbildung 5.11 Symboltabelle des Programms aus Abbildung 5.6

gramm der Abbildung 5.6. Die Symboltabelle zeigt alle Symbole, die im Programm definiert werden. Sie zeigt auch die Attribute zu jedem Symbol. Da der Assembler sehr stark Datentyp-orientiert ist, benötigt er diese Informationen und zeigt sie in der Listendatei für Ihren Gebrauch. Symbole werden in Labels, Variable und Zahlen unterteilt. Die Tabellen enthalten Werte, sofern diese zur Assemblierungszeit bekannt sind. Für jede Datenstruktur zeigt die Tabelle außerdem die jeweilige Länge an.

Querverweise

Die vom Assembler erstellte Querverweisdatei ist nicht sofort verwendbar. Sie müssen das CREF-Programm ausführen, um die .CRF-Datei in eine ASCII-Textdatei umzuwandeln. Dazu rufen Sie das CREF-Programm genauso auf wie den Assembler, mit der Ausnahme, daß Sie nur zwei Dateien spezifizieren: Die Eingabedatei mit einer .CRF-Erweiterung und eine Ausgabedatei mit einer .REF-Erweiterung. Wenn Sie das DOS-Kommando A>CREF eingeben, erhalten Sie eine Eingabeanforderung für die beiden Dateinamen. Als Alternative nimmt das Kommando A>CREF B:FIG5-10,B: als Eingabe die Datei B:FIG5-10.CRF und erstellt die Ausgabedatei B:FIG5-10.REF. In ähnlicher Weise können Sie mit dem Makroassembler verfahren. Genaueres dazu finden Sie im Makroassembler-Handbuch.

```
Figure 5.6   DOS Function Example

 Symbol Cross Reference                    (# is definition)        Cref-1

BAD_CLOSE_MSG. . . . . . . . . . .    50#    169
BAD_OPEN_MSG . . . . . . . . . . .    37#     86
BAD_READ_MSG . . . . . . . . . . .    46#    141
BAD_WRITE_MSG. . . . . . . . . . .    41#    105      159

CHANGE_RECORD. . . . . . . . . . .   126     128#
CHARACTER_LOOP . . . . . . . . . .    94#    110
CHAR_BAD_MSG . . . . . . . . . . .    60#    123
CLOSE_OK . . . . . . . . . . . . .   168     171#
CODE . . . . . . . . . . . . . . .     3#     28       28       28       173
CREATE_OK. . . . . . . . . . . . .    85      88#

DISK_TRANSFER_ADDRESS. . . . . . .    26#     68       95      144       145      146      148      152

ERROR_EXIT . . . . . . . . . . . .    76#     87      106      142       160      170

FCB. . . . . . . . . . . . . . . .     5#     71       82       99       136      155      165
FCB_BLOCK. . . . . . . . . . . . .    15#
FCB_CURRENT_RECORD . . . . . . . .    23#     89
FCB_DATE . . . . . . . . . . . . .    18#
FCB_DRIVE. . . . . . . . . . . . .     6#
FCB_EXT. . . . . . . . . . . . . .    11#
FCB_FILE_SIZE. . . . . . . . . . .    17#
FCB_NAME . . . . . . . . . . . . .     7#
FCB_RANDOM_RECORD. . . . . . . . .    24#     90       91      135
FCB_RECORD_SIZE. . . . . . . . . .    16#     92
FCB_RESV . . . . . . . . . . . . .    19#
FILE_ERROR_MSG . . . . . . . . . .    33#     75

INPUT_BAD_MSG. . . . . . . . . . .    54#    117

KEYBOARD_BUFFER. . . . . . . . . .    32#    112      115      124       150
KEYBOARD_ERROR . . . . . . . . . .   118#    130      132
KEYBOARD_LOOP. . . . . . . . . . .   111#    121      162
KEY_INPUT_OK . . . . . . . . . . .   116     122#

NO_FILE. . . . . . . . . . . . . .    74      80#

PROGRAM_EXIT . . . . . . . . . . .   127     163#
PROGRAM_START. . . . . . . . . . .    29      66#

RANDOM_READ_OK . . . . . . . . . .   140     143#
RANDOM_WRITE_OK. . . . . . . . . .   158     161#
RECORD_SIZE. . . . . . . . . . . .    31#     92       96      151
WRITE_OK . . . . . . . . . . . . .   104     107#
```

Abbildung 5.12 Querverweisliste des Programms aus Abbildung 5.6

Abbildung 5.12 zeigt die Ausgabe des Querverweis-Prozessors. Die Querverweisliste stammt aus dem Programm der Abbildung 5.6. Die linke Spalte enthält alle im Programm definierten Namen von Symbolen und Variablen. Gegenüber jedem Symbol befindet sich eine Reihe von Zahlen. Jede dieser Zahl gibt die Zeilennummer an, in welcher das Symbol erscheint. Wenn nach der Zeilennummer ein „#" steht, ist das Symbol in dieser Zeile definiert. Wenn „#" nicht erscheint, ist das Symbol in dieser Zeile nur angesprochen.

Wie können Sie die Querverweisliste benutzen? Der Querverweis läßt Sie die Verwendung jedes einzelnen Symbols bestimmen. Wenn z. B. eine Variable mit einem falschen Wert besetzt ist, identifiziert die Querverweisliste jeden Befehl, der diese Variable mit ihrem Namen anspricht. Dies soll dazu beitragen, die Befehle, die den Fehler verursachen könnten, zu bestimmen. Oder Sie modifizieren vielleicht gerade ein bereits bestehendes Programm. Dieses Programm wurde vielleicht von jemand anderem oder von Ihnen vor so langer Zeit geschrieben, daß Sie vergessen haben, wie es arbeitet. Wenn Sie nun eines der Unterprogramme ändern wollen, müssen Sie wissen, welche Teile des Programms dieses Unterprogramm benutzen. Die Querverweisliste zeigt jeden CALL-Befehl (oder auch jeden anderen dazu verwendeten Befehl), der dieses Symbol anspricht. Wenn Sie diese Befehle nun prüfen, können Sie ermitteln, ob die Änderung für alle Aufrufe des Unterprogrammes durchführbar ist. Der Querverweis macht die Arbeit des Ermittelns aller aufrufenden Befehle viel einfacher.

Linker

Die Ausgabe des Assemblers ist nicht ausführbar. Der vom Assembler erzeugte Maschinencode muß „gebunden" werden, bevor er ausgeführt werden kann.

Das auf der DOS-Diskette mitgelieferte LINK-Programm (LINK.EXE) führt eigentlich zwei verschiedene Funktionen aus. Es kann viele verschiedene Objektmodule zu einem einzigen Programm binden. Der Linker baut aber auch einen ausführbaren Lademodul aus einem Assemblerobjektmodul auf. Wir wollen diese Funktionen getrennt untersuchen.

Programme aus mehreren Modulen

Wie der Name anzeigt, ist es der Hauptzweck des LINK-Programms, mehrere Objektmodule zu einem einzigen ausführbaren Modul zu „linken" oder zu binden. Bisher waren alle Beispiele Programme aus jeweils einem einzigen Modul, das heißt, alle Funktionen waren in einem einzigen Quellprogramm enthalten. Es ist jedoch nicht immer möglich oder wünschenswert, so zu verfahren.

Es gibt mehrere Gründe dafür, warum ein Programm in mehrere Module zerlegt sein kann. Der erste ist die Größe. Ein sehr großes Programm wird sehr schwerfällig, schwierig zu editieren und zeitraubend zu assemblieren. Nehmen wir an, Sie machen einen Fehler in einer einzigen Zeile in einem Assemblerprogramm mit

5000 Zeilen. Sie müssen das gesamte Programm editieren, um diese eine Zeile zu ändern. Sie müssen dann das gesamte 5000-Zeilen-Programm assemblieren, was ziemlich viel Zeit in Anspruch nimmt. Nach einem verhältnismäßig kurzen LINK-Schritt ist das Programm dann endlich betriebsbereit.

Nehmen wir an, daß — anstelle eines 5000-Zeilen-Programmes — Sie das Programm in zehn Programmodule unterteilen, von denen jeder etwa 500 Zeilen lang ist. Um eine Einzelzeile zu editieren, müssen Sie nur eine 500-zeilige Quelldatei editieren. Das Assemblieren eines 500-Zeilen-Programms geschieht sehr viel schneller als das eines 5000-Zeilen-Programms. Der LINK-Schritt ist trotzdem verhältnismäßig schnell, auf jeden Fall im Vergleich zum Assemblieren des großen Programms. Beschneidet man die Größe des individuellen Moduls, läuft der Editier-/Assemblierprozeß also schneller ab.

Ein anderer Grund für die Unterteilung des Programmes in kleinere Module ist die Urheberschaft. An einem großen Programmprojekt arbeiten in der Regel mehr als eine Person. Wenn es dann nur eine einzige Quelldatei gibt, müssen die einzelnen Programmierer abwechselnd damit arbeiten. Dadurch wird alles sehr schnell schwerfällig.

Der letzte Grund für die Erstellung von modularen Programmen ist die Wiederverwendbarkeit. Wenn Sie ein Programm schreiben, verwenden Sie wahrscheinlich eine Anzahl von Unterprogrammen. Wenn Sie gut gearbeitet haben, erfüllt jedes dieser Unterprogramme eine spezielle Funktion mit einer gut dokumentierten Eingabe- und Ausgabespezifikation. Sie werden später ein anderes Programm schreiben und Sie möchten dann dasselbe Unterprogramm als Teil des neuen Programms benutzen. Wenn dieses Unterprogramm ein getrennter Programmmodul ist, ist das Anlinken an ein neues Programm problemlos. Sofern Sie keinen getrennten Modul aus ihm gemacht haben, müssen Sie es dann aus dem ursprünglichen Programm herauseditieren und in ein neues Programm einsetzen. Wenn Sie dies ein- oder zweimal probiert haben, werden Sie wahrscheinlich nach einem einfacheren Weg suchen.

Die Unterteilung eines Programmes in Module verlangt von Ihnen, dem Programmierer, einige Schritte. Sie müssen das Programm sorgfältig entwerfen, wenn Sie es in kleinere Komponenten zerlegen. Sie müssen die Parameter für Ein- und Ausgabe dieser kleineren Programme definieren. Und schließlich müssen Sie in der Lage sein, die Programmodule in Verbindung zueinander treten zu lassen. Die ersten beiden Schritte gehören zur Grundlage des Programmierens, und wir werden diese hier nicht besprechen. Der letzte Schritt ist Teil des Assemblers und Linkers, und wir wollen uns ansehen, wie er abläuft.

Wenn Sie ein Programm, in mehrere Module unterteilt, entworfen haben, muß das Erst- oder Hauptprogramm in der Lage sein, diese Unterprogramme aufzurufen. Der CALL-Befehl erledigt diese Aufgabe. Er hat einen einzigen Operanden, den Namen des Unterprogrammes. In allen bisherigen Beispielen war das Unterprogramm Teil desselben Programmoduls, so daß der Assembler genau wußte, welche Adresse das Unterprogramm bei der Ausführungszeit haben würde. Dadurch konnte der Assembler den richtigen Offsetwert für den Adressenteil des Befehls bestimmen.

Wir haben nun eine Situation, wo das Assemblieren des Unterprogramms getrennt vom Assemblieren des CALL-Befehls geschieht. Das bedeutet, daß der Assembler die richtige Adresse für den Aufruf nicht bestimmen kann. Da die Assemblierung des Unter- und des Hauptprogrammes getrennt erfolgen, besteht für den Assembler keine Möglichkeit, den richtigen Adresswert zu ermitteln. Aber das LINK-Programm kann diese Arbeit übernehmen. Der LINK-Schritt übernimmt die Bestimmung der Adressen. Alle Programmodule sind Teil der Eingabe an das LINK-Programm, so daß der Linker weiß, wo sich jedes Unterprogramm befindet. Der Linker kann dann die in den CALL-Befehlen verwendeten Adressen bestimmen, was der Assembler vorher nicht konnte.

EXTRN und PUBLIC

Doch der Linker kann das nicht alles allein machen. Der Programmierer muß dem Assembler sagen, auf welche Programme in einem anderen Programmodul zugegriffen wird. Dies geschieht mit einer PUBLIC-Anweisung, die dem Assembler sagt, daß dieses Symbol für andere Programme zur Verfügung steht. Der Programmierer sagt dem Assembler auch, welche Labels für diesen Programmodul extern sind. Die Assembleranweisung dafür ist EXTRN. Diese Anweisung erklärt das Label als extern für den aktuellen Assembliervorgang, so daß es der Assembler richtig behandeln kann. Der Assembler kennzeichnet jeden sich darauf beziehenden Befehl, so daß der Linker ihn später finden und die richtige Adresse einsetzen kann.

Die EXTRN-Anweisung dient zwei Zwecken. Sie sagt dem Assembler, daß das benannte Label extern für den aktuellen Assembliervorgang ist. Nun könnte der Assembler annehmen, daß jedes Label, das er während des Assemblierens nicht finden kann, extern ist. Das wäre in Ordnung, doch wenn Sie dann ein Label falsch geschrieben haben, so würde der Assembler annehmen, daß es sich um ein externes handelt, und keine Fehlermeldung ausgeben. Dadurch wird eine mögliche Fehlermeldung auf den LINK-Schritt verlagert. Dort würden Sie dann Ihren Schreibfehler feststellen. Für die meisten Benutzer kommt dies zu spät, und sie würden lieber früher darüber unterrichtet sein, besonders über etwas so Simples wie einen einfachen Schreibfehler. Deshalb gibt der Assembler eine Fehlermeldung für jedes Label aus, das er nicht finden kann.

Der zweite Grund für die EXTRN-Anweisung besteht darin, dem Assembler mitzuteilen, um was für eine Art von Label es sich handelt. Der Assembler ist an eine genaue Typeinteilung gebunden, er muß deshalb genau wissen, was jedes einzelne Symbol bedeutet. Erst dies läßt ihn die richtigen Befehle generieren. Für eine Variable kann die EXTRN-Anweisung ein Byte, ein Wort, ein Doppelwort oder eine andere Konstruktion kennzeichnen. Bezieht sich das Label auf ein Unterprogramm oder irgendein anderes Programm bzw. einen anderen Programmteil, kann es entweder NEAR oder FAR sein, je nachdem, in welchem Segment das Label sich befindet. Die EXTRN-Anweisung verlangt vom Programmierer, anzugeben, um was für einen Typ von Label es sich handelt. Da der Assembler auch das Adressieren der Segmente für das Programm ausführt, zeigt die EXTRN-Anweisung an, in welchem Segment das Label erscheint. Anstatt diese Information zu einem Teil der EXTRN-Syntax zu

```
The IBM Personal Computer Assembler 01-01-83          PAGE    1-1
Figure 5.13  Main routine

1                                          PAGE    ,132
2                                          TITLE   Figure 5.13  Main routine
3
4       0000                       STACK   SEGMENT STACK
5       0000      40 [                     DW      64 DUP(?)              ; Allocate space for the stack
6                        ????
7                             ]
8
9       0080                       STACK   ENDS
10
11      0000                       CODE    SEGMENT PUBLIC
12
13                                         EXTRN   OUTPUT_ROUTINE:NEAR, OUTPUT_CHARACTER:BYTE
14
15                                         ASSUME  CS:CODE
16
17      0000                       START   PROC    FAR
18
19      0000  1E                           PUSH    DS                     ; Segment of return address
20      0001  B8 0000                      MOV     AX,0                   ; Establish return address
21      0004  50                           PUSH    AX                     ; Offset of return
22      0005  FC                           CLD                            ; Ensure correct direction
23      0006  8C C8                        MOV     AX,CS                  ; Establish segment addressing
24      0008  8E D8                        MOV     DS,AX
25                                         ASSUME  DS:CODE                ; Indicate new segment addressing
26      000A  BE 001C R                    MOV     SI,OFFSET MESSAGE      ; Address of message string
27      000D                       LOOP:
28      000D  AC                           LODSB                          ; Get the next byte of the message
29      000E  A2 0000 E                    MOV     OUTPUT_CHARACTER,AL    ; Store in memory location
30      0011  E8 0000 E                    CALL    OUTPUT_ROUTINE         ; Output the character
31      0014  80 3E 0000 E 0A              CMP     OUTPUT_CHARACTER,10    ; Look for ending character
32      0019  75 F2                        JNE     LOOP                   ; Loop if not
33
34      001B  CB                           RET                            ; Return to DOS
35
36      001C  54 68 69 73 20 69    MESSAGE DB      'This is a test',13,10
37            73 20 61 20 74 65
38            73 74 0D 0A
39
40      002C                       START   ENDP
41      002C                       CODE    ENDS
42                                         END     START
```

Abbildung 5.13 Hauptprogramm

machen, wird sie durch die Lage der EXTRN-Anweisung bestimmt. Der Assembler nimmt an, daß sich ein externes Symbol in dem Segment befindet, in dem die EXTRN-Anweisung für dieses Symbol erscheint.

Abbildung 5.13 ist ein Beispiel für ein Assemblerprogramm, das den Gebrauch der EXTRN-Anweisung darstellt. Dieses Beispiel enthält zwei Labels, die für das Programm extern sind. Da gibt es als erstes eine Byte-Datenvariable, OUTPUT_CHARACTER. Die Angabe ":BYTE" nach dem Namen der Variablen ist der Weg, auf dem Sie das Attribut der Variablen spezifizieren. Das Programmlabel, OUTPUT_ ROUTINE ist als NEAR gekennzeichnet, um zu zeigen, daß es sich im selben Segment befindet. Das Programm in Abbildung 5.13 spricht diese Namen an, und der Assembler kann die richtigen Befehle erstellen. Wäre hier keine EXTRN-Anweisung vorhanden, würde der Assembler Fehler anzeigen. In der Assemblerliste können Sie „E" neben dem Adressenfeld der Befehle sehen, die externe Symbole ansprechen.

Wir müssen uns nun mit der anderen Seite des Problems beschäftigen. Woher weiß der Linker, wo er die externen Symbole auffindet? Abbildung 5.14 zeigt das Unterprogramm, das in Abbildung 5.13 angesprochen ist. Die Variablen und Labels, die sich auf ein anderes Programm beziehen, sind als PUBLIC erklärt. Das bedeutet, daß jeder andere Programmodul auf diese Werte zugreifen kann. Alle Variablen und Programmbezeichnungen im Programm, die nicht als PUBLIC erklärt sind, können von anderen Programmen nicht angesprochen werden. Obwohl dies als nachteilig

```
The IBM Personal Computer Assembler 01-01-83            PAGE    1-1
Figure 5.14  Output subroutine

1                                        PAGE    ,132
2                                        TITLE   Figure 5.14  Output subroutine
3
4        0000                    CODE    SEGMENT PUBLIC
5
6                                        ASSUME  CS:CODE,DS:CODE ; This must be true when called
7
8                                        PUBLIC  OUTPUT_CHARACTER,OUTPUT_ROUTINE
9
10       0000  ??                OUTPUT_CHARACTER        DB      ?
11
12       0001                    OUTPUT_ROUTINE  PROC    NEAR
13
14       0001  A0 0000 R                 MOV     AL,OUTPUT_CHARACTER     ; Get character to output
15       0004  B4 0E                     MOV     AH,14                   ; BIOS function
16       0006  BB 0000                   MOV     BX,0                    ; Set active page
17       0009  BA 0000                   MOV     DX,0                    ; BIOS parameters
18       000C  CD 10                     INT     10H                     ; Display routine
19       000E  C3                        RET                             ; Return to caller
20
21       000F                    OUTPUT_ROUTINE  ENDP
22       000F                    CODE    ENDS
23                                       END
```

Abbildung 5.14 Ausgabe-Unterprogramm

erscheinen mag, gäbe es doch ein anderes Problem, wenn alle Labels PUBLIC wären. Es würde bedeuten, daß jedes Label in jedem Modul, den Sie möglicherweise miteinbinden möchten, einzigartig sein müßte. Das heißt, daß Sie den gleichen Namen niemals zweimal benutzen könnten, nicht einmal in getrennten Modulen. Dies könnte die Wiederverwendbarkeit einiger Unterprogramme ernsthaft beeinträchtigen, denn sie könnten Jahre später benutzt werden. Alle Labels zu behalten und sicherzugehen, daß keines wiederholt wird, wäre eine schwierige Aufgabe. Beachten Sie, daß die PUBLIC-Anweisung keine zusätzlichen Attribute für die damit bezeichneten Symbole benötigt. Die normalen Assembleranweisungen berücksichtigen dies bereits.

Das LINK-Programm ordnet alle externen Symbole den entsprechenden PUBLIC-Erklärungen zu. Der Linker legt dann die korrekten Adresswerte in den Befehlen ab, die diese externen Werte ansprechen. Jedes Feld in der Assemblerliste, neben dem ein „E" steht, wird bearbeitet.

Der Linker verbindet außerdem alle Segmente mit dem gleichen Namen. In unseren Beispielen von Abbildung 5.13 und 5.14 befinden sich beide, das Haupt- und das Unterprogramm, im Segment mit dem Namen CODE. Da die EXTRN-Anweisung im Hauptprogramm angegeben hat, daß das Unterprogramm OUTPUT_ROUTINE NEAR war, wäre es besser im selben Segment gewesen. Die PUBLIC-Angabe in der SEGMENT-Anweisung sagt dem Linker jedoch, daß dieses Segment mit anderen Segmenten gleichen Namens verbunden werden kann. Damit kann der Linker die beiden Programmodule in das gleiche Segment binden.

Es gibt noch ein weiteres Segment in Abbildung 5.13, das wir besprechen sollten. Dieses Programm läuft als .EXE-Programm. Wenn DOS die Steuerung an ein .EXE-Programm übergibt, legt es einen Stack dafür an. Die Stackinformation kommt vom Linker, der sie im Header der .EXE-Datei abgelegt hat. Es ist Aufgabe des Programmierers, diesen Stack zu berücksichtigen. Tut er es nicht, gibt der Linker eine Warnung aus. Dies hindert das Programm unter normalen Umständen nicht an der Ausführung. Es beläßt das Programm jedoch mit dem Standardstack, welcher die

falsche Kapazität oder den falschen Platz besitzen könnte. Das Segment mit dem Namen STACK in Abbildung 5.13 berücksichtigt diese Notwendigkeit. Indem es den Namen STACK erhält und seine Attribut als STACK gesetzt wird, signalisiert es die Absicht, diesen Bereich als Stack zu benutzen. Der Linker sorgt auch dafür, daß der Stackpointer richtig gesetzt ist, wenn die Steuerung an das Programm übergeht.

Link-Vorgang

Lassen Sie uns nun die Schritte verfolgen, die diese Programmodule zu einem einzigen ausführbaren Modul verbunden haben. Sie assemblieren Programme mit den Kommandos, die im vorangegangenen Abschnitt besprochen wurden:

B>A:MASM FIG5-13,,,;
B>A:MASM FIG5-14,,,;

Dies erzeugt die beiden Objektmodule FIG5-13.OBJ und FIG5-14.OBJ. Sie rufen das LINK-Programm auf, um die Module zu binden. Abbildung 5.15 zeigt die Aktionen, die das LINK-Programm starten.

Dieses Beispiel nimmt an, daß die DOS-Diskette sich im Laufwerk A: befindet, die Datendiskette im Laufwerk B:, und daß das Laufwerk B: das voreingestellte Laufwerk ist. Nach dem Start fordert das LINK-Programm vom Benutzer die Namen der Objektdateien an, die zu linken sind. Sie geben die Dateinamen ohne die OBJ-Erweiterung ein. Wenn Sie mehr als einen Modul linken, geben Sie die einzelnen Namen jeweils durch ein „+" getrennt ein. Unser Beispiel linkt die Module FIG5-13 und FIG5-14.

```
A>LINK

IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982
Object Modules [.OBJ]: B:FIG5-13+B:FIG5-14
Run File [A:FIG5-13.EXE]: B:
List File [NUL.MAP]: B:
Libraries [.LIB]:

A>
```

Abbildung 5.15 Link-Lauf

Die Module werden in der gleichen Reihenfolge gelinkt, wie sie dem Linker angegeben werden. In diesem Falle steht der Code in FIG5-13 also vor dem Code in FIG5-14. Die Namen in umgekehrter Reihenfolge zu spezifizieren, würde in ähnlicher Weise ihre Reihenfolge im Code vertauscht haben. Normalerweise gibt es keinen Grund dafür, den Code in einer bestimmten Reihenfolge einzusetzen. Die einzige Ausnahme ist der Einsprungpunkt in das Programm. Nach diesem Beispiel werden wir uns darüber unterhalten, wie man diesen Einsprungpunkt behandelt.

Die nächste Eingabeanforderung des Linkers betrifft den Namen der Programmdatei. Der voreingestellte Name ist der Dateiname des ersten Objektmoduls mit einer Erweiterung .EXE. Sie können zwar den Dateinamen durch die Eingabe eines anderen Namens ändern, aber Sie können die .EXE-Erweiterung nicht verändern.

Die nächste Eingabeanforderung fragt nach dem Dateinamen, um die Linkliste zu speichern. Sie können für diese Datei irgendeinen Dateinamen wählen, standardmäßig wird jedoch keine Liste ausgegeben. Im Beispiel weist die Eingabe B: den Linker an, eine Linkliste auf dem Laufwerk B: abzulegen. Der Linker hat den Namen FIG5-13.MAP für die Datei gewählt. Abbildung 5.16 zeigt die Datei FIG5-13.MAP, die durch diese Linkoperation erstellt wurde. Wir werden gleich darauf zurückkommen.

Die letzte Eingabeanforderung vom Linker fragt nach Programmbibliotheken, um sie miteinzubeziehen. Für das Programmieren in einigen höheren Programmiersprachen kann es notwendig sein, daß Sie hier eine Laufzeitbibliothek benennen. Für unsere Assemblerprogramme brauchen wir eine solche Bibliothek nicht zu spezifizieren.

Linkliste

Abbildung 5.16 zeigt die Listenausgabe aus einem Linkschritt. An diesem einfachen Beispiel gibt es nicht viel zu betrachten. Es zeigt je eine Zeile für jedes Segment im Programm. Wenn wir quer über die Zeile gehen, finden wir die Start- und Endadressen für jedes Segment, so wie es dann in den Speicher geladen wird. Beachten Sie, daß das CODE-Segment eine Länge von 3CH hat. Dies entspricht den kombinierten Längen der CODE-Segmente in den beiden Programmodulen. Das andere Segment in unserem Beispiel ist das STACK-Segment mit einer Größe von 80H Bytes.

Es gibt einiges, das hinsichtlich der Adressen in der Linkliste zu beachten ist. Zunächst: Es handelt sich bei allen um 20-Bit Adressen und sie beginnen an der Stelle 0. Da DOS das Programm an eine andere Stelle als 0 laden wird, wird der Lader diese Werte neu bestimmen. Ihr relativer Wert bleibt jedoch gleich. Das andere, was bezüglich der Segmente zu beachten ist, ist der Umstand, daß sie nicht unmittelbar aufeinanderfolgend in den Speicher gebracht werden. Obgleich das CODE-Segment nur 3CH lang ist, liegt das STACK-Segment bei 40H. Segmente müssen an Paragraphengrenzen beginnen, damit die Offsetadressen korrekt blei-

```
 Start   Stop    Length  Name              Class
 00000H  0003EH  003FH   CODE
 00040H  000BFH  0080H   STACK

 Program entry point at 0000:0000
```

Abbildung 5.16 Linkliste für die Programme aus Abbildung 5.13 und 5.14

ben. Paragraphengrenzen lassen die Segmentregister direkt auf die erste Stelle eines Segmentes zeigen. So hat der Linker das STACK-Segment an die erste Paragraphengrenze nach dem Ende des CODE-Segmentes gesetzt. In diesem Falle, da CODE bei 3BH endet, ist die nächste Adresse, die durch 16 teilbar ist, 40H.

Vielleicht ist Ihnen aufgefallen, daß die kombinierte Länge der CODE-Segmente in den beiden Programmen nicht wirklich 3CH ist. Anstelle irgendeiner anderen Spezifikation hat der Linker den Beginn eines jeden CODE-Segmentteils an eine Paragraphengrenze gelegt. Der erste Modul, FIG5-13, hat eine Länge von 2BH. Der Linker setzt den folgenden Modul FIG5-14 an die nächste Paragraphengrenze, in diesem Fall 30H. Die Länge 0CH des zweiten Moduls ergibt die Gesamtlänge von 03CH für das CODE-Segment. Die Paragraphenausrichtung ist die Standardkombination von Assembler und Linker. Der Assemblerbefehl SEGMENT kann diese Ausrichtung auf entweder BYTE oder WORD ändern, wenn dies gewünscht wird. Die BYTE-Ausrichtung packt die Programmodule im Segment direkt zusammen. Dies ist die platzsparendste Methode, Module zu kombinieren. Die Paragraphenausrichtung garantiert jedoch, daß es während der Ausführung keine Probleme mit dem Adressieren der Segmente geben wird. Wenn ein Programm Adressberechnungen durchführt und nicht auf eine Paragraphengrenze ausgerichtet war, ist es möglich, daß Fehler auftreten.

Der letzte Teil der Linkliste zeigt den Einsprungpunkt für die Programmausführung. Diese Adresse ist, wie alle anderen, relativ zum Beginn des Moduls und wird durch den Lader neu bestimmt. Es gibt mehrere Wege, um die Startadresse für ein .EXE-Programm anzuzeigen. Ein Weg ist, das Programm beim ersten Byte des Programmoduls mit der Ausführung beginnen zu lassen. Dabei müssen Sie darauf achten, daß das erste Byte des ersten Segments in der erzeugten Datei auch der erste Befehl ist, den Sie ausgeführt haben wollen. Der bessere Weg, den Einsprungpunkt zu spezifizieren, ist, diesen in der END-Anweisung des Hauptprogrammes zu benennen. In Abbildung 5.13 lautet die letzte Programmanweisung

```
END     START
```

wobei START das Label für die erste auszuführende Instruktion ist. Da wir die Module in der richtigen Reihenfolge gelinkt haben, ist dies zufällig auch die erste Instruktion im Programm. Aber auch wenn wir die Module in der umgekehrten Reihenfolge hätten linken müssen, wäre der Einsprungpunkt dennoch korrekt adressiert. Versuchen Sie es.

In jeder Linkoperation sollte es nur eine END-Anweisung mit einer Startadresse geben. Beachten Sie, daß bei FIG5-14 kein Einsprungpunkt in der END-Anweisung benannt ist. Wenn mehr als ein Einsprungpunkt benannt sind, nimmt der Linker in der Regel den letztbenannten. Es ist allerdings besser, dies korrekt anzugeben als zu riskieren, daß der Linker eventuell den falschen Einsprungpunkt auswählt. Denken Sie daran, daß diese Methode, den Einsprungpunkt für ein Programm zu spezifizieren, nur für eine .EXE-Datei gilt. Ein .COM-Programm beginnt mit seiner Ausführung immer bei Offset 100H im Programmsegment.

DEBUG

Das DEBUG-Programm bietet Ihnen eine Möglichkeit, in einem Maschinenprogramm Fehler zu finden. Das DEBUG-Programm ermöglicht es Ihnen, sich Schritt für Schritt durch das Programm zu arbeiten und zu beobachten, was geschieht. DEBUG ist ein weiteres Programm, das auf der DOS-Diskette geliefert wird. Es wird wie jedes andere Programm geladen. Sie treten mit DEBUG in Verbindung, indem Sie die Tastatur und den Bildschirm benutzen. DEBUG fordert Befehle mit einem „—" an, wenn er eine Aktion von Ihnen erwartet.

Anstatt die Kommandos für DEBUG aufzulisten, wollen wir den Debugger benutzen, um die Arbeit des Programmes, das wir gerade in den Abbildungen 5.13 und 5.14 geschrieben haben, zu prüfen. Abbildung 5.17 zeigt die Auflistung dieses Vorgangs.

Dieses Beispiel ruft den DEBUG auf und benennt auch das Programm, mit dem gearbeitet wird, in diesem Falle FIG5-13.EXE. Nachdem DEBUG geladen ist, holt er sich das benannte Programm. DEBUG hat die Steuerung und zeigt mit dem Symbol „—" an, daß er eine Eingabe erwartet. Nichts geschieht mit dem Programm, bis Sie ihm sagen, was zu tun ist.

Das „R"-Kommando zeigt die Register, wie sie sind, wenn das Programm FIG5-13 geladen ist und die Steuerung erhält. Alle sind aus sich selbst verständlich, mit Ausnahme vielleicht der Flagwerte. Anstatt das Flagregister in hexadezimaler Schreibweise zu zeigen, stellt DEBUG die individuellen Flags dar. NV zeigt an: kein Überlauf, UP ist das Richtungsflag usw. Die letzte Zeile der Registeranzeige ist die nächste Instruktion, die auszuführen ist. An der Stelle 4C5:0000 befindet sich beispielsweise der Befehl PUSH DS. Wir sollten hier die Beschreibung des DEBUG-Programmes kurz unterbrechen, um die in den Registern dargestellte Information zu untersuchen. Die Register sind so belegt, wie sie es sein werden, wenn das Programm FIG5-13 die Steuerung vom Kommandoprozessor übernimmt. Beachten Sie, daß CS:IP auf den ersten Befehl zeigt, wie in der END-Anweisung des Assemblers bestimmt. Das DS- und ES-Register zeigen auf das Programmsegment-Präfix (PSP). Schließlich bestimmt SS:SP das STACK-Segment. Diese Registerbelegung steht im Gegensatz zur Belegung für eine .COM-Datei später in diesem Kapitel.

Um mehr von den Instruktionen zu sehen: „U" für deassemblieren zeigt die nächsten etwa 20 Instruktionen. Dies ist hilfreich, wenn Sie einen Code testen, für den Sie keine Auflistung haben. Das Deassemblieren des Codes läßt Sie die Instruktionen erkennen. Dies kann Ihnen Papier und Zeit ersparen, wenn Sie ein Programm etwas modifiziert haben. Ihre Auflistung deckt sich dann nicht mehr richtig mit dem tatsächlichen Programm. Das Deassemblieren des Programms hilft Ihnen dann, die korrekte Adresse für jede Instruktion zu bestimmen.

Doch hat die DEBUG-Deassemblierung im Vergleich mit der Auflistung mehrere Schwachstellen. Es gibt keine Kommentare (welche für das Verstehen des Programms wesentlich sein können) und die Speicherstellen werden nur durch die

```
B>A:DEBUG FIG5-13.EXE
-R

AX=0000  BX=0000  CX=003F  DX=0000  SP=0080  BP=0000  SI=0000  DI=0000
DS=04B5  ES=04B5  SS=04C9  CS=04C5  IP=0000   NV UP DI PL NZ NA PO NC
04C5:0000 1E            PUSH    DS

-U

04C5:0000 1E            PUSH    DS
04C5:0001 B80000        MOV     AX,0000
04C5:0004 50            PUSH    AX
04C5:0005 FC            CLD
04C5:0006 8CC8          MOV     AX,CS
04C5:0008 8ED8          MOV     DS,AX
04C5:000A BE1C00        MOV     SI,001C
04C5:000D AC            LODSB
04C5:000E A23000        MOV     [0030],AL
04C5:0011 E81D00        CALL    0031
04C5:0014 803E30000A    CMP     B,[0030],0A
04C5:0019 75F2          JNZ     000D
04C5:001B CB            RET     L
04C5:001C 54            PUSH    SP
04C5:001D 68            DB      68
04C5:001E 69            DB      69
04C5:001F 7320          JNC     0041

-D4C5:0

04C5:0000  1E B8 00 00 50 FC 8C C8-8E D8 BE 1C 00 AC A2 30   .8..P|.H.X>..,"0
04C5:0010  00 E8 1D 00 80 3E 30 00-0A 75 F2 CB 54 68 69 73   .h...>0..urKThis
04C5:0020  20 69 73 20 61 20 74 65-73 74 0D 0A 00 00 00 00    is a test......
04C5:0030  00 A0 30 00 B4 0E BB 00-00 BA 00 00 CD 10 C3 00   . 0.4.;..:..M.C.
04C5:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
04C5:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
04C5:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
04C5:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-RAX
AX 0000
:1234

-E 4C5:21
04C5:0021  69.    73.    20.    61.20   20.

-G 3C

AX=0E54  BX=0000  CX=003F  DX=0000  SP=007A  BP=0000  SI=001D  DI=0000
DS=04C5  ES=04B5  SS=04C9  CS=04C5  IP=003C   NV UP DI PL NZ NA PO NC
04C5:003C CD10          INT     10
-T

AX=0E54  BX=0000  CX=003F  DX=0000  SP=0074  BP=0000  SI=001D  DI=0000
DS=04C5  ES=04B5  SS=04C9  CS=F000  IP=F065   NV UP DI PL NZ NA PO NC
F000:F065 FB            STI
-T

AX=0E54  BX=0000  CX=003F  DX=0000  SP=0074  BP=0000  SI=001D  DI=0000
DS=04C5  ES=04B5  SS=04C9  CS=F000  IP=F066   NV UP EI PL ZR NA PE NC
F000:F066 FC            CLD
-G 4C5:3E

T
AX=0754  BX=0000  CX=003F  DX=0000  SP=007A  BP=0000  SI=001D  DI=0000
DS=04C5  ES=04B5  SS=04C9  CS=04C5  IP=003E   NV UP DI PL NZ NA PO NC
04C5:003E C3            RET
-G

his is  test

Program terminated normally
-R

AX=0754  BX=0000  CX=003F  DX=0000  SP=007A  BP=0000  SI=001D  DI=0000
DS=04C5  ES=04B5  SS=04C9  CS=04C5  IP=003E   NV UP DI PL NZ NA PO NC
04C5:003E C3            RET
-Q

B>
```

Abbildung 5.17 DEBUG-Sitzung für Abbildung 5.13 und 5.14

Adresse und nicht durch den Variablennamen identifiziert. Zum Beispiel erscheint die Anweisung bei 4C5:000E in Abbildung 5.13 als

MOV OUTPUT_CHARACTER,AL

während sie in der deassemblierten Auflistung als

```
MOV     [0030],AL
```

erscheint.

Beides stellt den gleichen Befehl dar. Der Variablenname OUTPUT_CHARACTER sagt dem Programmierer, der das Testen durchführt, mehr als die Speicheradresse 0030. Aber das DEBUG-Programm kennt die Variablennamen nicht und kann sich nur auf die Adressen stützen.

Das DEBUG-Programm produziert auch nicht unbedingt genau die Anweisungen, die dann wieder vom Assembler akzeptiert werden. Das heißt, daß einige Instruktionen anders aussehen. Die Anweisung bei 4C5:0014 deassembliert als

```
CMP     B,[0030],0A
```

Aber die Instruktion in Abbildung 5.13 ist tatsächlich

```
CMP     OUTPUT_CHARACTER,10
```

Der Deassembler arbeitet nur im Hexadezimalsystem, sowohl für die Eingabe als auch für die Ausgabe. Daher kommt das 0A. Wir haben bereits ausgeführt, warum wir [0030] erhalten, und nicht OUTPUT_CHARACTER. Aber was ist das „B,"?

Der Assembler arbeitet nur mit typisierten Variablen. Das heißt, daß die Variablen als Byte, Wort oder andere Arten von Variablen während der Assemblieroperation bekannt sind. Wenn also der Programmierer einen Direktbefehl eingibt, der den Speicher anspricht, weiß der Assembler, wie groß der Speicherplatz ist. In diesem Beispiel ist OUTPUT_CHARACTER als Byte-Variable bekannt. Aber das DEBUG-Programm hat keine Vorstellung davon, wie groß die Variable am Ort [0030] ist. Der Deassembler weiß jedoch, daß dieser Befehl ein Einzelbyte mit direkten Daten an den Ort [0030] überträgt. So bedeutet das „B,", daß die Direktoperation eine Bytebewegung ist. Um denselben Effekt im Assembler zu erreichen, wäre die Instruktion

```
CMP     BYTE PTR [0030],10
```

Sie können das „B," als Kürzel für BYTE PTR betrachten. Ähnlich steht „W," für WORD PTR, „L" zeigt einen langen oder FAR (return) Befehl an, usw.

Als Teil des Deassemblierens ist der Maschinencode für diesen Befehl dargestellt. An der Stelle 4C5:001C können Sie einige Instruktionen sehen, die nicht in Abbildung 5.13 erscheinen. Dies ist der Datenbereich, der den String „Dies ist ein Test." enthält. Das Deassemblierkommando weiß nämlich nicht, wann die Befehle enden und wo die Daten beginnen. So behandelt es alles als Befehl. (Übrigens, sollte sich Ihr Programm in diesen Datenbereich verirren, wäre dies die auszuführende Befehlsfolge.)

Das Ausgabekommado „D," kann Datenbereiche darstellen. Die Anzeige ist in zwei Teile unterteilt. Es gibt eine hexadezimale Auflistung der Speicherinhalte, anschließend deren ASCII-Darstellung. Befehle ergeben hier keinen Sinn, aber der Datenbe-

reich erscheint sehr klar. Wenn Sie einmal nichts Besseres zu tun haben, werden Sie vielleicht ein Programm zu schreiben versuchen, bei dem die Befehle die ASCII-Darstellung für Ihre Initialen bilden.

Der Debugger kann Register und Speicherinhalte modifizieren. Die Eingabe „R", gefolgt von einem Registernamen, zeigt das Register auf dem Bildschirm und bietet Ihnen eine Gelegenheit, den Wert zu ändern. Die Eingabe von „Return" läßt das Register unverändert, während die Eingabe eines neuen Wertes es verändert.

Sie können auch Speicherstellen modifizieren. Mit dem „E," für Editieren können Sie den Speicher modifizieren. DEBUG zeigt die Werte an den einzelnen Speicherstellen, gefolgt von einem „.". Sie können die Speicherstelle durch Eingabe eines neuen Wertes ändern, die Leertaste drücken, um auf die nächste Stelle weiterzugehen oder „Return" eingeben, um zur Kommandoeingabe zurückzukehren. In diesem Beispiel sind die ersten drei Stellen unverändert. Die Stelle 04C5:0024 wurde von 61H nach 20H verändert. Da es sich hier um den Datenbereich handelt, wird die zukünftige Meldung sich von der ursprünglich programmierten unterscheiden.

Alle Instruktionen, die Speicheradressen ansprechen, akzeptieren eine Adresse als Teil des Kommandos. Das „E"-Kommando zeigt die eingegebene Adresse genau wie das Ausgabekommando. Das Deassemblierkommando hätte ebenso eine Adresse benutzen können. Sie können die Adresse als Segment und Offset eingeben oder ganz einfach nur den Offset. Wenn Sie nur den Offset benutzen, spricht DEBUG das geeignete Segmentregister an. Für „U" benutzt er das CS-Segment. Für „D" und „E" zeigt das DS-Register auf das voreingestellte Segment.

Es ist nun Zeit für den Versuch, das Programm auszuführen. Wir könnten das Programm einfach starten und sehen, was passiert. Aber dafür brauchen wir kein DEBUG-Programm. Der Debugger erlaubt es uns, Stoppunkte in das Programm einzusetzen, die „Breakpoints" genannt werden. Durch das Einsetzen von Breakpoints in das Programm können wir das Programm zwingen, die Steuerung an DEBUG zurückzugeben. Dies gibt uns eine weitere Möglichkeit, die Register und den Speicher zu prüfen und den Programmablauf zu überwachen.

Das Laufkommando „G," überträgt die Steuerung von DEBUG an das Programm. Die Befehlsausführung beginnt an der Stelle, auf die das Registerpaar CS:IP zeigt (genau wie im richtigen Prozessor). Die Ausführung des getesteten Programms läuft weiter, bis ein Breakpoint erreicht wird. In unserem Beispiel haben wir einen Breakpoint an der Stelle 3CH eingesetzt. Da nur der Offset spezifiziert war, benutzt DEBUG den CS-Wert für das Segment. Ein Blick auf die Auflistung in Abbildung 5.14 zeigt, daß sich bei Offset 3CH der Befehl INT 10H befindet. Diese besondere Stelle wurde für das Beispiel gewählt, da dies ein Punkt ist, an dem die Steuerung an eine ROM BIOS-Routine übertragen wird. Durch die Überprüfung des Programms an diesem Punkt gehen wir sicher, daß wir die Register richtig gesetzt haben, bevor die BIOS-Routine ausgeführt wird.

Wenn das Programm den Breakpoint trifft, geht die Steuerung an DEBUG zurück. Er zeigt die Register zusammen mit dem nächsten Befehl, genau wie mit dem „R"-Kommando. Die Steuerung liegt wieder bei DEBUG, und Sie können irgendein Kommando eingeben.

Es gibt Grenzen für den Gebrauch von Breakpoints. Für einen Breakpoint wird der Operationscode 0CCH verwendet. Dieser Befehl veranlaßt ein INT 3. Dieser Softwareinterrupt gibt die Steuerung an DEBUG zurück. Da ein Befehl die Steuerung an DEBUG zurückgibt, muß der Breakpoint am Beginn eines solchen stehen. Wenn sich der Breakpoint irgendwo anders befindet, kann DEBUG die Steuerung nicht zurückerhalten, und das Programm wird einen anderen als den beabsichtigten Befehl ausführen. Wenn wir z. B. „-G 3D," spezifiziert hätten, wäre der Befehl bei 3CH ein INT 0CCH gewesen, und das Programm hätte sich irgendwo verlaufen.

Solange Breakpoints sorgfältig gesetzt sind, wird alles richtig laufen. Mit dem „G"-Kommando können Sie bis zu zehn Breakpoints setzen. Wenn einer von ihnen angesprochen wird, werden alle wieder auf ihren ursprünglichen Wert rückgesetzt. Ein Laufkommando ohne irgendwelche Breakpoints wird niemals auf einen der früher eingegebenen Breakpoints treffen, da diese alle entfernt worden sind. Wenn Sie mit der Ausführung beginnen und Ihr Programm stoppt oder läuft in eine endlose Schleife, ist es wahrscheinlich unmöglich, die Steuerung ohne den Gebrauch des System Reset (CTL-ALT-DEL) zurückzuerhalten, was bedeutet, daß Sie noch einmal von vorn beginnen müssen. Sie sollten also sehr vorsichtig sein, wenn Sie ein unbekanntes Programm starten.

Wenn Sie einen permanenten Breakpoint schaffen wollen, benutzen Sie „E", um das erste Byte eines Befehles in 0CCH zu ändern. Dieser Breakpoint bleibt dort für immer oder zumindest so lange, bis Sie ihn wieder ändern. Sie möchten vielleicht einen solchen Breakpoint am Eintrittspunkt einer Fehlerbearbeitungsroutine benutzen. Während des Programmtests wollen Sie die angetroffenen Fehler wahrscheinlich lieber selbst sorgfältig verfolgen, als ihre Bearbeitung dem Programm überlassen.

Es gibt auch noch eine andere Überlegung hinsichtlich der Breakpoints. Sie können einen Breakpoint nicht in einen Festwertspeicher einsetzen. Da Sie nicht in den ROM schreiben können, wird der Befehl 0CCH dort auch niemals gespeichert.

Das nächste DEBUG-Kommando ist „T". Dieses Trace-Kommando führt einen einzelnen Befehl des Programms aus. Unser Beispiel enthält mehrere Wiederholungen des „T"-Kommandos. Sie können sehen, daß das Programm die ersten Befehle der ROM BIOS-Routine, auf die INT 10H zeigt, ausführt. Die ROM BIOS-Routine ist, wie ihr Name ausdrückt, im ROM zu finden. Das Tracekommando ermöglicht es, das Programm zu „unterbrechen", während es Code aus dem ROM ausführt.

Das Tracekommando arbeitet, wenn das Tracebit im Flagregister gesetzt ist, bevor die Steuerung an das Benutzerprogramm geht. Dieses Tracebit verursacht eine Unterbrechung (INT 1) nach der Ausführung eines jeden Befehls. Der INT 1-Vektor gibt die Steuerung an DEBUG zurück. Die INT 1-Routine löscht automatisch das Tracebit. Das bedeutet, daß DEBUG nicht nach jedem seiner eigenen Befehle unterbrochen wird. Der Tracebefehl ist ein sehr guter Weg, um sich durch einen schwierigen Codeabschnitt hindurchzuarbeiten. DEBUG zeigt jeden Befehl mit den Registerinhalten unmittelbar vor seiner Ausführung. Da dabei Interrupts anstelle von Breakpoints benutzt werden, können wir den Programmablauf selbst im ROM verfolgen.

Zurück zum Beispiel. Das „-G 4C5:3E"-Kommando erlaubt es dem ROM BIOS-Unterprogramm, vollständig abzulaufen. Beachten Sie, daß dieses Unterprogramm ein „T" auf den Bildschirm geschrieben hat. Die ROM BIOS-Routine für Interrupt 10H gibt nämlich Zeichen auf den Bildschirm aus. Und „T" ist das erste Zeichen unserer Meldung. Da wir nun ziemlich sicher sind, daß das Programm richtig ausgeführt wird, läßt die Eingabe von „G" ohne Breakpoints das Programm jetzt vollständig ablaufen.

Da es sich bei diesem Beispiel um eine .EXE-Datei gehandelt hat, konnte INT 20H nicht benutzt werden, um die Steuerung an DOS zurückzugeben. Stattdessen hat das Programm das DS-Register und einen 0-Wert in den Stack geschoben. Der FAR-Return-Befehl am Ende des Hauptprogramms gibt die Steuerung an DOS zurück. DEBUG erkennt dies und holt sich die Steuerung am Ende des getesteten Programms zurück. Hätte es sich um ein .COM-Programm gehandelt, hätte INT 20H in gleicher Weise die Steuerung zu DEBUG zurückgebracht. Nachdem wir uns mit diesem Beispiel genug beschäftigt haben, können wir DEBUG verlassen und zu DOS zurückgehen. Dazu dient das „Q"-Kommando.

Umwandlung von .EXE in .COM

Auf der DOS-Diskette gibt es ein Dienstprogramm mit Namen EXE2BIN. Dieses Programm wandelt ein Programm von der .EXE-Darstellung in einen .COM-Typ um. Das EXE2BIN-Programm kann jedoch nicht für alle Programme verwendet werden. Hier zeigen wir nun eine Methode, wie durch die Verwendung von DEBUG jedes Programm in eine .COM-Datei umgewandelt werden kann.

```
The IBM Personal Computer Assembler 01-01-83            PAGE    1-1
Figure 5.18  .EXE to .COM Example


1                                              PAGE    ,132
2                                              TITLE   Figure 5.18  .EXE to .COM Example
3       0000                           CODE    SEGMENT
4       0100                                   ORG     100H
5                                              ASSUME  CS:CODE, DS:CODE
6
7       0100  BA 0109 R                        MOV     DX,OFFSET MESSAGE
8       0103  B4 09                            MOV     AH,9                ; DOS Display string
9       0105  CD 21                            INT     21H                 ; Print the string
10      0107  CD 20                            INT     20H                 ; Return to DOS
11
12      0109  54 68 69 73 20 69        MESSAGE DB      'This is a test',10,13,'$'
13            73 20 61 20 74 65
14            73 74 0A 0D 24
15      011A                           CODE    ENDS
16                                             END
```

Abbildung 5.18 Beispielumwandlung .EXE zu .COM

Abbildung 5.18 zeigt das Programm, das wir umwandeln wollen. Dieses Programm führt genau die gleiche Funktion aus, wie das vorhergehende Beispiel. Das heißt, es gibt „Dies ist ein Test" auf den Bildschirm aus. Doch wird hier dazu die Funktion 9 von INT 21H verwendet.

Beachten Sie, daß das Programm als eine .COM-Datei geschrieben wurde. Ausschlaggebend ist die Anweisung ORG 100H vor dem ersten Befehl. Der Rest des

Programmes muß über das Code-Segment verschiebbar sein. Dies ist kein Trick, der nur für dieses einfache Beispiel anwendbar ist, sondern Sie sollten sich dies merken für den Fall, daß Sie ein Programm schreiben, das Sie in .COM umwandeln wollen.

```
B>A:ASM FIG5-18,,,;
The IBM Personal Computer Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Warning Severe
Errors  Errors
0       0

B>A:LINK FIG5-18,,,;
IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982

 Warning: No STACK segment
There was 1 error detected.

B>RENAME FIG5-18.EXE FIG5-18.COM

B>A:DEBUG
-NFIG5-18.COM
-L
-M 400 1000 100

-U100 10F

06D7:0100 BA0901        MOV     DX,0109
06D7:0103 B409          MOV     AH,09
06D7:0105 CD21          INT     21
06D7:0107 CD20          INT     20
06D7:0109 54            PUSH    SP
06D7:010A 68            DB      68
06D7:010B 69            DB      69
06D7:010C 7320          JNC     012E
06D7:010E 69            DB      69
06D7:010F 7320          JNC     0131
-D100

06D7:0100  BA 09 01 B4 09 CD 21 CD-20 54 68 69 73 20 69 73   :..4.M!M This is
06D7:0110  20 61 20 74 65 73 74 0A-0D 24 00 00 00 00 00 00    a test..$......
06D7:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
06D7:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
06D7:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
06D7:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
06D7:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
06D7:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-RCX
 CX 0380
:120

-W

Writing 0120 bytes
-Q

B>DEBUG FIG5-18.COM
-R

AX=0000  BX=0000  CX=0120  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=04B5  ES=04B5  SS=04B5  CS=04B5  IP=0100   NV UP DI PL NZ NA PO NC
04B5:0100 BA0901        MOV     DX,0109

-Q

B>FIG5-18
This is a test
```

Abbildung 5.19 Beispielumwandlung .EXE zu .COM

Sie assemblieren und linken das Programm nun auf normale Weise. Aber bevor Sie das DEBUG-Programm starten, benennen Sie die .EXE-Datei in eine .COM-Datei um. Sie müssen dies tun, denn mit DEBUG können Sie keine .EXE-Datei schreiben. Abbildung 5.19 zeigt die Reihenfolge der Schritte, die zu befolgen sind. Das Beispiel gibt das DEBUG-Kommando ohne den Dateinamen aus. Sie könnten auf dieser Zeile FIG5-18.COM spezifizieren, aber wenn dies unterbleibt, können wir einige andere DEBUG-Funktionen vorführen. Mit dem „N"-Kommando des DEBUG

können Sie eine Datei benennen. Das „L"-Kommando lädt diese Datei in den Speicher. Wenn der Dateiname in der DEBUG-Kommandozeile erscheint, erfüllt er dieselbe Funktion wie das „N"- und „L"-Kommando.

Mit der jetzt geladenen Datei finden Sie das Programm bei Offset 400H. Das „M"-Kommando verschiebt den Speicherblock von 400H auf 100H. Die Länge von 1000H wurde gewählt, um sicherzugehen, daß sie für das Programm ausreicht. Das Programm ist nun im .COM-Dateiformat und kann auf die Diskette zurückgeschrieben werden. Aber bevor dies geschieht, modifizieren Sie das CX-Register, damit es die tatsächliche Länge des Programms enthält. Für alle Lese- und Schreiboperationen auf Disketten enthält das CX-Register im DEBUG die Länge der Datei. Nachdem die .COM-Datei viel kürzer ist als die .EXE-Datei, können wir Platz auf der Diskette sparen, indem wir CX auf den richtigen Wert für unser Programm setzen. Das „W"-Kommando schreibt die Datei nun zurück auf die Diskette. Dies ist, nebenbei bemerkt, ein weiterer Vorteil des Gebrauchs von .COM-Dateien. DEBUG schreibt keine .EXE-Datei auf die Diskette, da die Headerinformation nicht mehr im Speicher steht. DEBUG kann aber .COM-Dateien auf Diskette schreiben. Wenn Sie ein Programm testen und lieber ein oder zwei Bytes modifizieren wollen, als das Programm neu zu assemblieren (dies nennt man „patching"), können Sie so verfahren. Modifizieren Sie lediglich das Programm und überzeugen Sie sich, daß CX richtig gesetzt ist. Schreiben Sie dann mit „W" das Programm auf die Diskette zurück.

Kommando	Beschreibung
D	Speicherinhalt ausgeben
E	Speicherinhalt ändern
F	Speicherblock auffüllen
G	Programm ausführen
H	Addieren und Subtrahieren hexadezimal
I	Eingabeport lesen und ausgeben
L	Laden von Diskette
M	Speicherblock verschieben
N	Dateinamen festlegen
O	Zeichen auf Ausgabeport ausgeben
Q	DEBUG beenden
R	Register ausgeben
S	Suchen nach einem bestimmten Byte-String
T	Einzelbefehle ausführen
U	Befehlscode deassemblieren
W	Daten auf Diskette schreiben

Abbildung 5.20 DEBUG-Kommandos

Wir wollen DEBUG nun verlassen und uns mit der neuen Version von FIG5-18.COM befassen. Ein Blick auf die Register zeigt deren Belegung für eine .COM-Datei. Im Gegensatz hierzu die in Abbildung 5.17 gezeigte Registerbelegung einer .EXE-Datei. Diese Unterschiede sollen helfen, einige der Unterschiede zwischen .COM- und .EXE-Dateien herauszuarbeiten.

Es gibt noch weitere Kommandos für DEBUG. In Abbildung 5.20 können Sie alle DEBUG-Kommandos sehen. Und im DOS-Handbuch finden Sie eine detaillierte Erläuterung dafür.

6 Eigenschaften des Makro-Assemblers

Dieses Kapitel wird einige Eigenschaften des IBM Makro-Assemblers erklären. Obgleich wir alle Instruktionen des 8088-Prozessors besprochen haben, gibt es noch weitere Befehle, die Teil des Assemblers sind. Wir haben bereits einige dieser Pseudo-Instruktionen, wie die Datenbestimmungsoperatoren DB und DW, behandelt. In diesem Kapitel werden noch leistungsfähigere Assembleroperationen vorgestellt. Was sie miteinander verbindet, ist die Fähigkeit, das Schreiben von Assemblerprogrammen einfacher und leichter zu machen.

Wir wollen in diesem Kapitel zwei hauptsächliche Bereiche besprechen. Der erste ist die Makro-Operation des Makro-Assemblers. Makros sind leistungsstarke, befehlserstellende Werkzeuge. Für Kapitel 7, welches den Arithmetikprozessor 8087 behandelt, ist es notwendig, etwas von Makros zu verstehen. Der zweite Bereich behandelt Datenstrukturen. Wir haben bisher die Datendefinition in Verbindung mit Datentypen besprochen. Hier wollen wir nun Datenstrukturen definieren, die aus Bytes und Wörtern zusammengesetzt sind. Wir befassen uns außerdem mit Segmenten, Strukturen und Datensätzen zur Datendefinition.

Makros

Ein Makro ist ein Programmierwerkzeug, das es Ihnen ermöglicht, Ihre eigenen Assembleroperationen zu schaffen. In Wirklichkeit ist ein Makro ein Text-Austauschmechanismus. Der Makroprozessor läßt Sie einen neuen Operationscode für den Prozessor definieren. Als Teil der Definition geben Sie dem Assembler den Text für diesen Operationscode. Wenn der Assembler auf diesen neu definierten Operationscode stößt, bezieht er sich auf die gesicherte Definition des Makros. Er setzt den Text dieser Definition in die Assemblierung ein. Ein Programm kann z. B. eine oft wiederholte Reihenfolge von Befehlen als Makro definieren. Immer wenn diese Instruktionen im Program erscheinen sollen, kann der Programmierer stattdessen dann den Makro-Operationscode benutzen.

Es gibt zwei unterschiedliche Schritte im Gebrauch eines Makros. Im ersten Schritt definiert das Programm den Makro. Der Programmierer gibt dem Makro einen Namen und eine Definition. Die Definition besteht aus den Assembleroperationen und Befehlen, die erstellt werden, sooft der Makroname erscheint. Der zweite Schritt ist das Aufrufen des Makros. Dies erfolgt, wenn der Assembler auf den als Operationscode benutzten Makronamen trifft. Der Assembler setzt dann die definierten Befehle an die Stelle des Makronamens.

Lassen Sie uns ein Beispiel aus den Instruktionen des 8087 benutzen, das wir im Kapitel 7 besprechen wollen. Es gibt ein Problem beim Schreiben von Programmen, die die Instruktionen für den Arithmetikprozessor 8087 benutzen. Der Makro-Assembler kann die Opcodes des 8087 nicht erzeugen. Um den 8087 zu benutzen, müssen Sie die 8087-Befehle selbst formulieren, indem Sie entweder die DB-Operatoren oder WAIT- und ESC-Opcodes verwenden. Am besten macht man dies mit

einem Satz von Makros, die es Ihnen ermöglichen, Instruktionen für den 8087 zu schreiben. Ein Programm kann dann die Instruktionen des 8087 spezifizieren, obwohl sie nicht Teil des normalen Befehlsvorrats sind.

Ein Makro wird meistens beim Programmieren in Assembler benutzt. Obgleich es keinen Grund gibt, warum eine höhere Programmiersprache keinen Makroprozessor benutzen könnte, findet man sie nicht oft. Der IBM PC Makro-Assembler bietet jedoch die Möglichkeit, mit Makros zu arbeiten. Wie Sie gesehen haben, gibt es zwei Versionen des Assemblers. Der kleine Assembler, ASM, sieht keine Möglichkeit vor, Makros zu verwenden. Der große Assembler MASM ermöglicht alle Makrooperationen, die in diesem Kapitel besprochen werden. Für die Benutzung von MASM muß Ihr PC einen Speicher von mindestens 96K Bytes haben.

Ein sehr einfacher Makro, den Sie für einen 8087-Operationscode benutzen können, ist der FENI-Makro. Der 8088 Makro-Assembler erkennt den FENI-Operationscode, der eigentlich eine an den 8087 gerichtete Anweisung ist, nicht. Die Abbildung 6.1 zeigt die beiden Schritte im Makroprozeß: Die Definition des FENI-Makros und sein späterer Aufruf. Abbildung 6.1 enthält zwei Teile: Teil (a) ist die Quelldatei für das Programm, während Teil (b) die Assemblerliste des Programms enthält. Abbildung 6.1 listet die beiden Teile getrennt auf, um zu zeigen, welche Teile durch den Programmierer geschrieben wurden und welche der Makroprozessor erstellt hat.

Ein Programm definiert einen Makro mit dem MACRO-Operationscode. In Abbildung 6.1 sieht die Makro-Definition wie folgt aus

```
FENI    MACRO
        ;--- Körper des Makros
        ENDM
```

Die MACRO-Anweisung ist ein Pseudobefehl. Dieser spezielle Befehl sagt dem Assembler, daß ein Makro definiert werden soll. Das Namensfeld des Befehls enthält den Namen, den wir dem Makro geben, in diesem Fall FENI. Nach dieser Kopfzeile folgen die Befehle, die später den Makronamen ersetzen sollen. Zum Schluß sagt der ENDM-Operationscode dem Assembler, daß das Ende des Makros erreicht ist. Der Text zwischen MACRO und ENDM bildet den Körper des Makros. In Abbildung 6.1 ist der Hauptteil des FENI-Makros ein DB-Operator. Da es keine 8088-Instruktion gibt, die sich mit dem FENI-Befehl deckt, muß der Maschinencode für FENI über DB-Operatoren aufgebaut werden.

Es ist zu beachten, daß kein Maschinencode während der Definition eines Makros erstellt wird. Wir sehen dies auch daran, daß die Adress- und Datenspalten der Assemblerliste leer sind. Wenn der Assembler die Makrodefinition zum ersten Mal sieht, speichert er die Definition für späteren Gebrauch weg. Später ruft das Programm der Abbildung 6.1 den FENI-Makro auf. Der Programmierer benutzt den Makronamen FENI wie einen normalen Assemblerbefehl, etwa CLD oder DAA. Der Assembler spricht nun seine gesicherte Definition des FENI-Makros an. Er nimmt den Text aus dem Hauptteil des Makros und setzt ihn am Punkt FENI in die Assemblierung ein. Das „+"-Zeichen, das links vor der DB-Anweisung in der Assemblerliste erscheint, ist eine Anzeige, daß der Makroprozessor diese Code-

```
        PAGE    ,132
        TITLE   Figure 6.1  Macro Definition

FENI    MACRO
        DB      0DBH,0E0H
        ENDM

CODE    SEGMENT
        ASSUME  CS:CODE

        FENI
CODE    ENDS
        END
```

Figure 6.1(a) Source File for Program

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.1  Macro Definition

1                                           PAGE    ,132
2                                           TITLE   Figure 6.1  Macro Definition
3
4                                   FENI    MACRO
5                                           DB      0DBH,0E0H
6                                           ENDM
7
8       0000                        CODE    SEGMENT
9                                           ASSUME  CS:CODE
10
11                                          FENI
12      0000  DB E0           +             DB      0DBH,0E0H
13
14      0002                        CODE    ENDS
15                                          END
```

Figure 6.1(b) Assembly Listing for Program

Abbildung 6.1 Makrodefinition: (a) Quellprogram; (b) Assemblerliste des Programms

zeile eingesetzt hat. Sie können außerdem das Quellprogramm mit der Assemblerliste vergleichen. Sie werden feststellen, daß die Quelldatei nur den FENI-Befehl zeigt. Die Assemblerliste zeigt jedoch den FENI-Operationscode mit dem nachgestellten Hauptteil des FENI-Makros. In diesem Falle ist der Körper des Makros nur die DB-Zeile.

Dieses einfache Beispiel macht die Leistungsfähigkeit des Makro-Prozessors deutlich. Wir brauchten einen Befehl mit Namen „FENI", der nicht vom Assembler erkannt wurde. Ohne die Makroeigenschaften müßte ein Programmierer die Operation

DB 0DBH,0E0H

für jeden FENI-Befehl kodieren. Über Makros können wir einmal den FENI-Makro definieren und ihn dann ständig als Befehl benutzen. Es gibt zwei gute Gründe für die Benutzung von Makros. Erstens ist es damit leichter, ein Programm zu schreiben, und zweitens sagt es viel mehr aus, den Namen FENI zu lesen, als die Zeichenfolge DB 0DBH,0E0H.

Sie können einen Makro mit einem Unterprogramm vergleichen. Ein Unterprogramm ist ein Codeabschnitt, der nur an einer Stelle des Programmes definiert ist. Das Programm kann die Steuerung von jedem Ort innerhalb des Programmes an das Unterprogramm abgeben. Die Benutzung eines Unterprogrammes erspart dem Programmierer Zeit und Platz im Programm. Anstatt die Befehle eines Unterprogramms jedesmal zu wiederholen, wenn es gebraucht wird, rufen Sie das Unterprogramm nur auf. Es führt dann die definierte Funktion aus und die Steuerung kehrt zum Aufrufer zurück.

In ähnlicher Weise definiert das Assemblerprogramm einen Makro nur an einer Stelle. Ist er definiert, können Sie ihn von jeder Stelle im Assemblerprogramm auf-

rufen. Die Benutzung eines Makros erspart dem Programmierer Zeit und Platz in der Quelldatei. Anstatt die Befehle eines Makros jedesmal zu wiederholen, wenn sie erscheinen sollen, rufen Sie einfach den Makro auf. Der Assembler fügt die definierten Befehle ein und fährt fort, den nächsten Befehl in der Quelldatei zu bearbeiten.

Der Unterschied zwischen Makro und Unterprogramm liegt in der Zeit, zu der sie gebraucht werden. Ein Makro ist eine Textbearbeitungsoperation. Ein Makro wird definiert und „ausgeführt" zum Zeitpunkt der Assemblierung. Die Ausführung eines Makros ist der Austausch des Makronamens gegen den Text des Makrokörpers. Das Unterprogramm wird dagegen während des Assemblierens definiert, aber nicht ausgeführt, bevor das Programm läuft. Wir sagen, daß ein Makro während des Assemblierens ausgeführt wird, während ein Unterprogramm während des Programmlaufs ausgeführt wird.

Am besten unterscheidet man einen Makro von einem Unterprogramm, indem man sich vergegenwärtigt, wann ihre Wirkung eintritt. Der Makroprozessor ist nicht notwendigerweise Teil einer Programmiersprache. Nehmen wir an, Sie seien ein Rechtsanwalt und würden viele Testamente schreiben. Da Testamente meistens sehr ähnlich sind, könnten Sie einen Satz Testamentmakros definieren, welcher alle üblichen Teile der Testamente, die Sie schreiben, enthält. Der erste Teil des Testaments wäre spezifisch, er würde die Namen der in einem Testament berücksichtigten Parteien enthalten. Das restliche Testament würde aus verschiedenen Testamentmakros bestehen, die die üblichen Abschnitte eines Testaments enthalten. Wenn der „Testamentprozessor" die Ausführung übernimmt, ist das Ergebnis ein Textdokument. Die Makros werden dabei zu den allgemeinen Passagen des Testamentes erweitert. Sie müssen lediglich noch die jeweils spezifischen Textabschnitte zwischen die Makros einfügen.

Wenn Makros und Unterprogramme sich in sehr vieler Hinsicht so ähnlich sind, warum benützt man dann lieber einen Makro als ein Unterprogramm? In vielen Fällen ist es möglich, beide zu benutzen. Sie können eine Reihe von Befehlen entweder als Makro oder als Unterprogramm definieren. Wenn Sie diese Befehlsreihe in einem Programm brauchen, können Sie den Makro oder das Unterprogramm aufrufen. Welches Sie wählen, hängt von Ihrer Definition der Befehlsfolge ab. Zeit- und Platzüberlegungen bestimmen die Entscheidung. In den meisten Fällen ergibt der Gebrauch eines Makros ein größeres Programm, das heißt, man braucht für dieselbe Funktion mehr Bytes für den Maschinencode. Der Code, der Makros benutzt, ist jedoch schneller in der Ausführung. Es gibt keinen Aufwand für das Ausführen eines Unterprogrammaufrufs bzw. die Rückkehr, wenn die Befehlsfolge durchlaufen wird. Bei einem kleinen Programm sollten Sie also ein Unterprogramm benutzen. Für ein superschnelles Programm verwenden Sie besser Makros.

Der FENI-Makro in Abbildung 6.1 ist eine ganz offensichtlich gute Wahl für einen Makro. Nicht nur, daß der Code als Makro schneller abläuft als in der Form eines Unterprogramms, er ist auch noch kürzer. Die CALL-Instruktion für ein NEAR-Unterprogramm erfordert drei Bytes. Der FENI-Makro braucht nur zwei. Für die 8087-Makros wären mehr Bytes für den Maschinencode notwendig, wenn man den 8087 über Unterprogrammaufrufe versorgen würde, als dies beim Gebrauch der Makros

der Fall ist. Auch erreichen wir durch die Verwendung der Makros eine optimale Ausführungsgeschwindigkeit.

Argumente für Makros

Es gibt eine Eigenschaft der Unterprogramme, die man auch bei der Makrobearbeitung findet. Es ist möglich, die Aktionen eines Makros durch den Gebrauch von Argumenten zu modifizieren. Genau wie die Argumente für ein Unterprogramm dessen Ausführung modifizieren können, bestimmen die Argumente eines Makros die echten Befehle, die erstellt werden. Und genau wie ein Unterprogramm ohne Argumente relativ wenig Gebrauchsmöglichkeiten bietet, so verhält es sich auch mit einem Makro ohne Argumente.

Betrachten wir ein anderes, einfaches Beispiel. Wir haben bestimmt, daß unser Programm an einer spezifischen Speicherstelle an vielen Orten im Programm einen anderen konstanten Wert hinzufügt. Anstatt die Befehle

```
ADD     MEMORY_BYTE,5
```

oder

```
ADD     MEMORY_BYTE,7
```

viele Male zu schreiben, würden wir besser einen Makro für diesen Befehl benutzen. Aber der konstante Wert ist bei jedem Gebrauch dieses Befehls unterschiedlich. So machen wir den konstanten Wert zu einem Argument für den Makro. Abbildung 6.2 zeigt die Definition und den Aufruf des ADDBYTE-Makro. Das Beispiel benutzt das Symbol CONSTANT in der Makrodefinition als Argument für den Makro. Alle Symbole, die im Operandenfeld der MACRO-Anweisung erscheinen, werden als Argumente behandelt. Zum Zeitpunkt der Definition des Makros hat das Symbol CONSTANT keine Bedeutung. Es hält lediglich den Platz im Text des Makros frei. Später, wenn der Makro aufgerufen wird, setzt die Textbearbeitung des Makroassemblers den Wert des Arguments an die Stelle des Symbols, das während der Definition des Makros benutzt wurde.

```
The IBM Personal Computer MACRO Assembler 01-01-83           PAGE    1-1
Figure 6.2  Macro Argument

 1                                                   PAGE    ,132
 2                                                   TITLE   Figure 6.2  Macro Argument
 3
 4                                          ADDBYTE MACRO    CONSTANT
 5                                                  ADD      MEMORY_BYTE,CONSTANT
 6                                                  ENDM
 7
 8        0000                              CODE    SEGMENT
 9                                                  ASSUME   CS:CODE
10
11        0000  ??                          MEMORY_BYTE      DB       ?
12
13        = 0004                            FOUR             EQU      4              ; Symbol for constant
14
15                                                  ADDBYTE           2
16        0001  2E: 80 06 0000 R 02     +           ADD      MEMORY_BYTE,2
17                                                  ADDBYTE           4
18        0007  2E: 80 06 0000 R 04     +           ADD      MEMORY_BYTE,4
19                                                  ADDBYTE           FOUR
20        000D  2E: 80 06 0000 R 04     +           ADD      MEMORY_BYTE,FOUR
21
22        0013                              CODE    ENDS
23                                                  END
```

Abbildung 6.2 Argumente für einen Makro

Es ist wichtig, zu beachten, daß das Makroargument ein Textargument ist. Da der Makroprozessor tatsächlich ein Textprozessor ist, kennt er Zahlen und Buchstaben nicht auseinander. Dies erlaubt es dem Makroaufruf, anstelle der Zahl ein Symbol zu benutzen. Die Auswertung dieses Zeichenstrings geschieht durch den Assembler, nicht durch den Makroprozessor. Der Makroprozessor setzt den Textstring des Makroaufrufs an die Stelle des Symbols, welches in der Makrodefinition benutzt wurde. Auf diese Weise ist der Wert FOUR für das Programm genauso verwendbar wie die Konstante „4".

Die Fähigkeit, Symbole als Argumente für einen Makro zu benutzen, ist entscheidend für das nächste Beispiel. Dieser Makro, einer der 8087-Befehle, erfordert ein Argument, das mit großer Sicherheit im normalen Gebrauch ein Symbol ist. Der Makro FLDCW ist eine 8087-Instruktion, die eine Speicheradresse benötigt. Da wir in unseren Programmen die meisten Speicheradressen mit Symbolen angesprochen haben, wollen wir auch mit den 8087-Befehlen weiter so verfahren.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.3  FLDCW Macro

1                                              PAGE    ,132
2                                              TITLE   Figure 6.3  FLDCW Macro
3
4                                      FLDCW   MACRO   SOURCE
5                                              DB      09BH
6                                              ESC     0DH,SOURCE
7                                              ENDM
8
9       0000                           CODE    SEGMENT
10                                             ASSUME  CS:CODE
11
12      0000  ????                     MEMORY_LOCATION DW      ?
13
14                                             FLDCW   MEMORY_LOCATION
15      0002  9B                   +           DB      09BH
16      0003  2E: D9 2E 0000 R     +           ESC     0DH,MEMORY_LOCATION
17                                             FLDCW   ES:[DI]
18      0008  9B                   +           DB      09BH
19      0009  26: D9 2D            +           ESC     0DH,ES:[DI]
20                                             FLDCW   MEMORY_LOCATION[BX+SI]
21      000C  9B                   +           DB      09BH
22      000D  2E: D9 A8 0000 R     +           ESC     0DH,MEMORY_LOCATION[BX+SI]
23
24      0012                           CODE    ENDS
25                                             END
```

Abbildung 6.3 FLDCW-Makro

Die Abbildung 6.3 zeigt den FLDCW-Makro und mehrere seiner Aufrufe. Beachten Sie, daß FLDCW das Symbol SOURCE als ein Argument benutzt. SOURCE ist die Adresse, von welcher der 8087 das Steuerwort lädt. FLDCW benutzt die 8088 ESC-Instruktion, um den erforderlichen Maschinencode zu erstellen. Die ESC-Instruktion braucht aber einen Adresswert, um das mod-r/m Byte für den Befehl zu bestimmen.

Der Makro benutzt das Argument SOURCE, um genau das zu tun. Diese Anordnung des FLDCW-Makros ermöglicht eine sehr natürliche Programmiermethode. In der gleichen Weise, wie Sie

```
INC      MEMORY_LOCATION
```

schreiben würden, können Sie die 8087-Instruktion

```
FLDCW    MEMORY_LOCATION
```

schreiben. Dies funktioniert nicht nur für Speicheradressen, die mit Symbolen arbeiten, sondern auch für die anderen Adressmodi. Abbildung 6.3 enthält mehrere

Beispiele für Methoden von Basis- und Indexadressierung zur Bestimmung der Speicheroperanden. Da der Makroprozessor das Argument als ein Stück Text behandelt, kann das Argument aus jeder gewünschten Textkette gebildet werden.

Sie können einen Makro mit mehr als einem Argument definieren. Die einzige Begrenzung für die Anzahl der Argumente, die Sie bei einem Makro verwenden können, ist die Anzahl der Symbole, die Sie in eine einzelne Assemblerzeile setzen können. Der Makroprozessor behandelt alles, was nach der MACRO-Anweisung steht, als Argument. Sie benutzen Kommas, um die Symbole während der Makrodefinition voneinander zu trennen. Eine MACRO-Anweisung mit drei Argumenten sieht aus wie folgt:

EXAMPLE MACRO ARG1, ARG2, ARG3

In ähnlicher Weise müssen Sie beim Aufruf eines Makros einen Wert für jedes Argument des Makros spezifizieren. Sollten Sie ein Argument auslassen, setzt der Assembler stattdessen eine Textkette mit der Länge 0 ein. Manchmal ist dies erwünscht, aber oft entsteht dadurch falscher Befehlscode. Wenn es mehr als ein Argument für einen Makro gibt, trennen die Kommas im Makroaufruf den Argumenttext. Dies ist genau der gleicher Weg, auf dem Sie mehrere Parameter in jeder beliebigen 8088-Anweisung spezifizieren würden, weshalb dies für Sie nicht schwierig sein sollte. Ein Aufruf für unseren vorgenannten Makro mit drei Argumenten könnte sein:

EXAMPLE 5,[BX],MEMORY_BYTE

Unser nächstes Beispiel wird Ihnen einige Möglichkeiten für mehrfache Argumente zeigen.

Bedingte Assemblierung

Bis hierher haben die besprochenen Makros Unterprogramme nachgeahmt, sowohl was die Operation wie auch den Gebrauch der Argumente betrifft. Der nächste Schritt ist die bedingte Bearbeitung. Genau wie das Unterprogramm die Fähigkeit hat, seinen Ablauf in Abhängigkeit von den Bedingungen zum Zeitpunkt der Ausführung zu ändern, sollte der Makro in der Lage sein, die Codegenerierung in Abhängigkeit von den Bedingungen zum Zeitpunkt der Assemblierung zu ändern.

Der IBM-Makroassembler kann bedingtes Assemblieren ausführen. Bedingtes Assemblieren ist aber nicht notwendigerweise Teil des Makroarbeitsablaufes. Wir können bedingtes Assemblieren überall im Programm benutzen. Am häufigsten erscheint bedingtes Assemblieren jedoch bei Makros. Nur der Makroassembler MASM führt bedingtes Assemblieren auf dem IBM PC durch.

Wie bei Makros wird bedingtes Assemblieren während der Assemblierung und nicht während der Ausführung des Programms durchgeführt. Bedingtes Assemblieren gestattet dem Programmierer, den Assembler so zu „programmieren", daß er verschiedene Befehlsfolgen assembliert. Der Assembler bestimmt, was assembliert wird, indem er einen während des Assemblierens bekannten Parameter benutzt. Obwohl wir diese Fähigkeit zu jedem Zeitpunkt während der Assemblierung benutzen können, wollen wir sie in erster Linie in Hinblick auf das Assemblieren von Makros betrachten.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.4 Conditional Macro Assembly

 1                                          PAGE    ,132
 2                                          TITLE   Figure 6.4 Conditional Macro Assembly
 3
 4                                  FIDIVR  MACRO   SOURCE
 5                                          IFE     2-TYPE SOURCE
 6                                                  DB      09BH            ;; FWAIT
 7                                                  ESC     037H,SOURCE     ;; FIDIVR word
 8                                          ENDIF
 9                                          IFE     4-TYPE SOURCE
10                                                  DB      09BH            ;; FWAIT
11                                                  ESC     017H,SOURCE     ;; FIDIVR short integer
12                                          ENDIF
13                                          ENDM
14
15      0000                        CODE    SEGMENT
16                                          ASSUME  CS:CODE,DS:CODE
17
18      0000  ????                  TWO_BYTE        DW      ?
19      0002  ????????              FOUR_BYTE       DD      ?
20      0006  ??                    ONE_BYTE        DB      ?
21
22                                          FIDIVR  TWO_BYTE
23      0007  9B                    +               DB      09BH
24      0008  DE 3E 0000 R          +               ESC     037H,TWO_BYTE
25
26                                          FIDIVR  FOUR_BYTE
27      000C  9B                    +               DB      09BH
28      000D  DA 3E 0002 R          +               ESC     017H,FOUR_BYTE
29
30                                          FIDIVR  ONE_BYTE
31
32      0011                        CODE    ENDS
33                                          END
```

Abbildung 6.4 Bedingte Makro-Assemblierung

Abbildung 6.4 zeigt eine bedingte Assemblierung während der Generierung des 8087-Befehlsmakros FIDIVR. Dieser Makro erfordert wegen der verschiedenen Variablentypen bedingtes Assemblieren. Sie können den Befehl FIDIVR auf zwei unterschiedliche Operandentypen anwenden, wie wir in Kapitel 7 noch sehen werden. Der Operand kann eine ganze Zahl mit entweder zwei oder vier Bytes sein. Wir möchten gerne, daß der Assembler den korrekten, dem Typ des Operanden entsprechenden Maschinencode erzeugt. Wie wir gesehen haben, hat der ADD-Befehl tatsächlich mehrere Formen, entsprechend den Operanden, die an den Assembler geliefert werden. Der Assembler bestimmt dann entsprechend den Operanden die korrekte Form des ADD-Befehls. Wir möchten das gleiche mit dem FIDIVR-Befehl ausführen können. Aber nun muß der Makroprozessor die Art des Operanden bestimmen und den korrekten Befehl erstellen.

Der FIDIVR-Befehl kann einen von zwei unterschiedlichen Operanden haben, und entsprechend diesem Operanden ist die sich ergebende Instruktion unterschiedlich. So sollte unsere Makroerweiterung für FIDIVR den richtigen Operanden zeigen. Dies ist durch zwei Dinge möglich: Bedingtes Assemblieren und die TYPE-Angabe.

Der Assembler verfügt über eine TYPE-Anweisung, die als Ergebnis die Länge des Operanden zurückgibt. Für das FIDIVR-Beispiel nehmen wir an, daß der Operand eine ganze Zahl mit entweder zwei oder vier Bytes ist. Deshalb ist das Resultat der TYPE-Anweisung zwei bzw. vier.

Der Ausdruck

    IFE   2-TYPE SOURCE

im FIDIVR-Makro prüft den TYPE des Operanden SOURCE. Der arithmetische Ausdruck 2-TYPE SOURCE ergibt 0, wenn der Operand SOURCE eine ganze Zahl von zwei Byte Länge ist, und einen Wert ungleich 0 für jeden anderen Operandentyp. Die Anweisung IFE (Assemble if Equal) sagt dem Assembler, die Ausdrücke nach der IFE-Anweisung zu assemblieren, wenn das Operandenfeld der IFE-Anweisung gleich 0 ist. Dies ist der Fall, wenn der Operand SOURCE eine ganze Zahl von zwei Byte Länge ist. Wenn der IFE-Ausdruck „wahr" ergibt, werden alle Ausdrücke nach der IFE-Anweisung assembliert, bis eine ENDIF-Anweisung angetroffen wird. Für unser Beispiel bedeutet dies, daß der Code

```
DB      09BH
ESC     37H,SOURCE
```

assembliert wird, wenn der Operand SOURCE eine ganze Zahl von zwei Byte Länge ist. Der erste Makroaufruf in Abbildung 6.4 zeigt, daß der Makro eine ganze Zahl von zwei Byte Länge als Operand benutzt. So wählt der Assembler die Zeichenfolge ESC 37H für die Makroerweiterung.

Da die FIDIVR-Instruktion den beiden unterschiedlichen Operanden entsprechend zwei Optionen hat, benutzt der Makro eine zweite IFE-Klausel für die andere Bedingung. Wenn der Operand eine ganze Zahl von vier Byte Länge ist, erstellt der Makro den Code für ESC 17H. Die Makroerweiterungen in Abbildung 6.4 zeigen die beiden unterschiedlichen Codegenerierungen.

Beachten Sie, daß der letzte Makroaufruf in Abbildung 6.4 einen Operanden hat, welcher keines der beiden Kriterien erfüllt. Da keine der beiden IFE-Anweisungen „wahr" ergibt, wird keine von ihnen assembliert. In diesem Fall erstellt der Makroprozessor keinerlei Text.

Es gibt unterschiedliche Prüfungen, die der Assembler mit der IF-Klausel vornehmen kann. Die Tabelle in Abbildung 6.5 zeigt diese verschiedenen Prüfungen. Das allgemeine Format der IF-Anweisung ist:

```
IFxx    Ausdruck
...
ELSE
...
ENDIF
```

IF Befehl		Erzeuge Code, wenn
IF	Ausdruck	Ausdruck nicht gleich 0
IFE	Ausdruck	Ausdruck gleich 0
IFDEF	Symbol	Symbol als extern erklärt ist
IFNDEF	Symbol	Symbol nicht als extern erklärt ist
IFB	<Argument>	Argument leer ist
IFNB	<Argument>	Argument nicht leer ist
IFIDN	<Arg1>,<Arg2>	String Arg1 identisch mit String Arg2 ist
IFDIF	<Arg1>,<Arg2>	String Arg1 ungleich String Arg2 ist
IF1		der Assembler in Pass 1 ist
IF2		der Assembler in Pass 2 ist

Abbildung 6.5 IF-Ausdruck für bedingtes Assemblieren

Der Assembler bearbeitet den Code, der nach der IFxx-Anweisung steht, wenn die Bedingung „wahr" ergibt. Der einer IF-Anweisung folgende assemblierte Code wird entweder durch eine ELSE-Anweisung oder eine ENDIF-Anweisung beendet. Die ELSE-Anweisung ist optional. Erscheint sie, wird der nachfolgende Code assembliert, wenn die Bedingung der IF-Anweisung nicht erfüllt ist. Die ENDIF-Anweisung beendet das bedingte Assemblieren und muß erscheinen.

Lassen Sie uns in einem weiteren Beispiel einige andere Anwendungen von bedingtem Assemblieren betrachten. Das Beispiel in Abbildung 6.6 zeigt den Gebrauch einer anderen IF-Klausel, IFB. Es zeigt auch den Gebrauch von verschachtelten Bedingungen. Der Makro ist hier FLD, die Ladeinstruktion für den 8087. Dieser Befehl erfordert mehrere Bedingungen in seiner Assemblierung, da er in den folgenden Formen auftreten kann.

```
FLD
FLD  1
FLD  short_real        (4 Bytes)
FLD  long_real         (8 Bytes)
FLD  temporary_real    (10 Bytes)
```

Das Operandenfeld des FLD-Makros kann entweder leer, eine Konstante, eine 4-Byte-Variable, eine 8-Byte-Variable oder eine 10-Byte-Variable sein. Der FLD-Makro muß bestimmen, was davon der Fall ist und den richtigen Code erstellen (Kapitel 7 erklärt diese Datentypen in allen Einzelheiten).

Der IFB-Ausdruck legt fest, ob der Operand vorhanden ist oder nicht. Ist der Operand nicht vorhanden, erstellt der Assembler den richtigen Code für diesen Fall, da IFB „wahr" ergibt. Der erste Makroaufruf zeigt dies, welcher den Code

```
DB   09BH,0D9H,0C1H
```

erstellt.

Die in diesem Abschnitt der IF-Klausel enthaltene EXITM-Anweisung ist der Befehl zur Beendigung der Makroverarbeitung. Immer wenn der Assembler auf diese Anweisung trifft, während er einen Makro bearbeitet, beendet er die Makroverarbeitung so, als hätte die Anweisung ENDM gelautet. Bei diesem Makro läßt sie den Assembler den verbleibenden Teil des Makros überspringen. Das Verlassen des Makros auf diese Weise läßt auf der Assemblerliste die Warnung „Open conditionals: 1" erscheinen. Diese Meldung macht Sie darauf aufmerksam, daß der Assembler die ENDIF-Anweisung, die zur bearbeiteten IF-Anweisung eigentlich gehört, nicht getroffen hat. Dies passiert, weil Sie den Makro vorzeitig verlassen haben. Obwohl dies nicht erwünscht ist, richtet es doch keinen Schaden an. Befindet sich die EXITM-Klausel nicht in einer bedingten Anweisung, so erscheint keine Warnung.

EXITM ist in diesem Makro notwendig, da der Assembler alle bedingten Anweisungen prüft, selbst wenn er sie nicht assembliert. In diesem Fall, wenn der Operand SOURCE leer ist, verhindert die ELSE-Klausel, daß irgendein anderer Fall von FLD generiert werden kann. Der Assembler fährt jedoch fort und evaluiert die Anweisung

```
IFE     TYPE SOURCE
```

```
The IBM Personal Computer MACRO Assembler 01-01-83            PAGE    1-1
Figure 6.6  Nested Conditional Assembly

1                                           PAGE    ,132
2                                           TITLE   Figure 6.6  Nested Conditional Assembly
3
4                                   FLD     MACRO   SOURCE
5                                           IFB     <SOURCE>
6                                                   DB      09BH,0D9H,0C1H          ;; FLD ST(1)
7                                                   EXITM
8                                           ELSE
9                                                   IFE     TYPE SOURCE
10                                                          DB      09BH,0D9H,0C0H+SOURCE   ;; FLD ST(i)
11                                                  ENDIF
12                                                  IFE     4 - TYPE SOURCE
13                                                          DB      09BH
14                                                          ESC     8,SOURCE        ;; FLD short_real
15                                                  ENDIF
16                                                  IFE     8 - TYPE SOURCE
17                                                          DB      09BH
18                                                          ESC     40,SOURCE       ;; FLD long_real
19                                                  ENDIF
20                                                  IFE     10 - TYPE SOURCE
21                                                          DB      09BH
22                                                          ESC     01DH,SOURCE     ;; FLD temporary_real
23                                                  ENDIF
24                                          ENDIF
25                                          ENDM
26
27      0000                        CODE    SEGMENT
28                                          ASSUME  CS:CODE, DS:CODE
29
30      0000  ????????                  FOUR_BYTE       DD      ?
31      0004  ????????????????          EIGHT_BYTE      DQ      ?
32      000C  ??????????????????        TEN_BYTE        DT      ?
33            ??
34
35                                          FLD
36      0016  9B D9 C1              +               DB      09BH,0D9H,0C1H
37                                          FLD     1
38      0019  9B D9 C1              +                       DB      09BH,0D9H,0C0H+1
39                                          FLD     FOUR_BYTE
40      001C  9B                    +                       DB      09BH
41      001D  D9 06 0000 R          +                       ESC     8,FOUR_BYTE
42                                          FLD     EIGHT_BYTE
43      0021  9B                    +                       DB      09BH
44      0022  DD 06 0004 R          +                       ESC     40,EIGHT_BYTE
45                                          FLD     TEN_BYTE
46      0026  9B                    +                       DB      09BH
47      0027  DB 2E 000C R          +                       ESC     01DH,TEN_BYTE
48
49      002B                        CODE    ENDS
50                                          END
Open conditionals: 1
```

Abbildung 6.6 Verschachtelte bedingte Assemblierung

obgleich er keinerlei Code erstellen kann. Wenn SOURCE leer ist, gibt der Assembler jedoch eine Syntax-Fehlermeldung aus. Sie können den Fehler ignorieren, aber es widerspricht unserem Prinzip, eine Assemblierung mit Fehlermeldungen zu akzeptieren. Der Gebrauch von EXITM dagegen erzeugt eine Warnung „Open conditionals". Obgleich diese Warnung nicht erwünscht ist, ist sie das kleinere der beiden Übel.

Beachten Sie, daß der FLD-Makro die ELSE-Klausel benutzt, um anzuzeigen, daß der Assembler die Befehle zur Auswertung des Operandenfeldes nur dann ausführen soll, wenn das Operandenfeld nicht leer ist. Die IF-Anweisungen mit dem TYPE-Operator bestimmen, welcher Operandentyp für den Makroaufruf benutzt wurde. Obgleich im Makro-Assembler Manual nicht erwähnt, gibt der TYPE-Operator 0 zurück, wenn der Operand eine Konstante und kein Symbol ist. Unser Makro hat diesen Weg aus Gründen der besseren Bearbeitung und nicht der Eleganz gewählt.

Wiederholungsmakros

Der Makro-Assembler verfügt über einige spezielle Makroformen, wenn Sie denselben Codeabschnitt mehrere Male wiederholen wollen. Diese Makroausdrücke sind REPT, IRP und IRPC. Jeder dieser Ausdrücke arbeitet selbst wie ein Makro und erstellt den nachfolgenden Code, bis der Assembler auf die ENDM-Anweisung trifft.

Um eine Befehlsfolge ganz einfach zu wiederholen, benutzen Sie den REPT-Makro.

```
REPT    Ausdruck
; . . . REPT Makrokörper
ENDM
```

Dadurch wird der Code im Makro-Hauptteil dupliziert. Der Wert des Ausdrucks bestimmt die Anzahl der Textwiederholungen.

Sie können verschiedene Argumente für jede Wiederholung mit dem IRP-Makro benutzen.

```
IRP     dummy,<Liste>
;... IRP Makrokörper
ENDM
```

In diesem Fall wird der Hauptteil des Makros für jeden Eintrag in der Parameterliste gesondert erzeugt. Bei jedem Durchgang durch den Makrohauptteil ersetzt der Assembler den Parameter „dummy" durch den nächsten Wert aus der Liste. Die Einträge der Liste müssen numerische Ausdrücke sein.

Wenn Sie in der Liste Zeichenwerte verwenden wollen, benutzen Sie den IRPC-Makro.

```
IRPC    dummy,Zeichenkette
;... IRPC Makrokörper
ENDM
```

Hier wird für jedes Zeichen des Strings einmal der Makrohauptteil durchlaufen. Der Parameter „dummy" wird bei jedem Durchlauf durch das nächste Zeichen in der Kette ersetzt. Abbildung 6.7 zeigt ein Beispiel für jede der erwähnten Wiederholungsfunktionen.

MACRO-Operatoren

Das IRPC-Beispiel in Abbildung 6.7 zeigt auch den Gebrauch des „&"-Symbols. Dieser Makro-Operator verbindet zwei Datenwörter miteinander. In unserem Beispiel verbindet der „&"-Operator den Parameter CHAR mit dem konstanten Text „X". Wie Sie sehen können, ergibt dies einen gültigen Registernamen.

Eine andere wertvolle Makrofunktion ist die LOCAL-Anweisung. Die LOCAL-Anweisung definiert ein Label, das nur im Makro benutzt wird. Dieses Label muß für jeden

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.7  Repeat Macros

1                                          PAGE    ,132
2                                          TITLE   Figure 6.7  Repeat Macros
3
4        0000                    CODE      SEGMENT
5                                          ASSUME  CS:CODE, DS:CODE
6
7                                          REPT    3              ; Repeat the code 3 times
8                                          INC     AX
9                                          ENDM
10       0000  40              +           INC     AX
11       0001  40              +           INC     AX
12       0002  40              +           INC     AX
13
14                                         IRP     VALUE,<5,10,15,20>
15                                         ADD     AX,VALUE
16                                         ENDM
17       0003  05 0005         +           ADD     AX,5
18       0006  05 000A         +           ADD     AX,10
19       0009  05 000F         +           ADD     AX,15
20       000C  05 0014         +           ADD     AX,20
21
22                                         IRPC    CHAR,ABCD
23                                         ADD     AX,CHAR&X
24                                         ENDM
25       000F  03 C0           +           ADD     AX,AX
26       0011  03 C3           +           ADD     AX,BX
27       0013  03 C1           +           ADD     AX,CX
28       0015  03 C2           +           ADD     AX,DX
29
30       0017                    CODE      ENDS
31                                         END
```

Abbildung 6.7 Wiederholungsmakros

Makroaufruf eindeutig sein. Nehmen wir an, Sie wollen einen Makro, der einen Codeabschnitt erzeugt, wie folgt schreiben:

```
AAAAA:  ADD   AL,[BX]
        INC   BX
        LOOP  AAAAA
```

Beim ersten Aufruf des Makros geht alles glatt. Aber wenn Sie den Makro im gleichen Programm zum zweiten Mal benutzen, erscheint auch das Label AAAAA zum zweiten Mal. Der Assembler kann keine zwei Label gleichen Namens in einem Programm zulassen und kennzeichnet dies als einen Fehler.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.8  Use of LOCAL in macros

1                                          PAGE    ,132
2                                          TITLE   Figure 6.8  Use of LOCAL in macros
3
4                                PAUSE     MACRO   TIME
5                                          LOCAL   LABEL
6                                          MOV     CX,TIME
7                                LABEL:    LOOP    LABEL
8                                          ENDM
9
10       0000                    CODE      SEGMENT
11                                         ASSUME  CS:CODE
12
13                                         PAUSE   100
14       0000  B9 0064         +           MOV     CX,100
15       0003  E2 FE           + ??0000:   LOOP    ??0000
16
17                                         PAUSE   1000
18       0005  B9 03E8         +           MOV     CX,1000
19       0008  E2 FE           + ??0001:   LOOP    ??0001
20
21       000A                    CODE      ENDS
22                                         END
```

Abbildung 6.8 Verwendung von LOCAL in Makros

Wenn Sie das Label AAAAA als LOCAL im Makro deklarieren, ist das Problem gelöst. Der Assembler setzt seinen eigenen eindeutigen Namen für jedes Vorkommen des Labels AAAAA. Wenn der Assembler ein LOCAL-Symbol zum ersten Mal sieht, gibt er ihm den Namen „??000". Für das nächste Vorkommen lautet der Name „??0001" usw. Jedes Label ist eindeutig und einmalig in der Assemblierung, so daß es keine Fehler geben kann. Abbildung 6.8 zeigt den Gebrauch der LOCAL-Anweisung. Dieser Makro, PAUSE, setzt einen Schleifenzähler und läuft dann durch diese Schleife, wobei er ein lokales Label benutzt. Der Makro läßt ein Programm für eine variable Zeitspanne pausieren. Wenn Sie LOCAL in einem Makro brauchen, muß es als erste Anweisung im Makro erscheinen, unmittelbar nach der MACRO-Anweisung.

Symbol	Bedeutung
;;	**Kommentar zur Verwendung nur innerhalb des Makros**
&	**Verknüpfen Text mit Parameter**
!	**Übernehmen nächstes Zeichen direkt**
%	**Umwandeln Ausdruck in Zahlenwert**

Abbildung 6.9 Makrosymbole

Es gibt spezielle Symbole, die Ihnen helfen, Makros und ihre Parameter zu handhaben. Die Tabelle in Abbildung 6.9 zeigt Ihnen die vier Symbole zusammen mit ihren Bedeutungen.

Abbildung 6.10 ist die Assemblerliste eines Programms, das diese Symbole benutzt. Wir haben bereits den Gebrauch von „;;" in einigen der 8087-Makros gesehen. Dieses spezielle Kommentarfeldkennzeichen weist den Makroprozessor an, das Kommentarfeld während der Makroverarbeitung zu übergehen. Dadurch können Sie Kommentare in einen Makro einsetzen, ohne daß diese bei jedem Makroaufruf erscheinen. Auch den Gebrauch von „&" haben wir in Abbildung 6.7 gesehen.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.10 Macro Special Characters

1                                          PAGE     ,132
2                                          TITLE    Figure 6.10 Macro Special Characters
3
4        = 0000                    VALUE   EQU      0
5
6                                  EXAMPLE MACRO    PARAMETER
7                                          DB       'MSG&PARAMETER'    ;; Comment appears only in definition
8                                          INC      AX
9                                          ENDM
10
11
12       0000                      CODE    SEGMENT
13                                         ASSUME   CS:CODE
14
15                                         EXAMPLE %VALUE
16       0000  4D 53 47 30        +        DB       'MSG0'
17       0004  40                 +        INC      AX
18
19       0005                      CODE    ENDS
20                                         END
```

Abbildung 6.10 Makrosonderzeichen

Das „!"-Symbol erlaubt es Ihnen, jedes beliebige Zeichen als nächstes Zeichen zu verarbeiten. Sie werden diese Operation brauchen, wenn Sie eines der speziellen Makrosymbole, z. B. „%" im Makro verwenden wollen, ohne damit eine Makro-Operation auszulösen. Schließlich wandelt der „%"-Operator ein Symbol in die Zahl um, die es gegenwärtig darstellt. Sie können diese spezielle Eigenschaft dazu benutzen, Dinge während der Makroverarbeitung zu numerieren. In unserem Beispiel numeriert der Makro Meldungen entsprechend der Zahl, die durch VALUE dargestellt wird.

INCLUDE-Anweisung

Die INCLUDE-Assembleranweisung fügt einen Text aus einer anderen Datei in das Quellprogramm ein. Die INCLUDE-Anweisung ist im besonderen anwendbar auf einen Satz von Makros, wie z. B. auf einen Satz von 8087-Makros. Die 8087-Makros enthalten alle 8087-Befehle.In jedes Programm, das den 8087 verwendet, müssen Sie einen solchen Satz von Makros oder eine Teilmenge davon einbeziehen. Aber es ist unsinnig, eine Kopie dieser Makros in jede Quelldateieinzufügen. Außerdem nehmen die Makros soviel Raum ein, daß der Quellcode Ihre Diskette sehr schnell füllen würde, wenn Sie versuchen sollten, eine Kopie der Makros in jede Datei einzufügen.

Die INCLUDE-Funktion des Assemblers regelt das Problem. Die Anweisung

INCLUDE Dateiname

nimmt den Inhalt der benannten Datei und fügt ihn in das Quellprogramm ein. Der Assembler setzt den Dateiinhalt an der Stelle der INCLUDE-Anweisung ein. Die INCLUDE-Anweisung ist höchst geeignet für die Arbeit mit Makrobibliotheken, wie z.B. mit den 8087-Makros. Sie setzen die INCLUDE-Anweisung an den Beginn des Programms und jede 8087-Instruktion im Programm wird korrekt übersetzt.

Ähnlich können Sie die INCLUDE-Anweisung benutzen, um andere Programmabschnitte in Ihr Programm einzufügen. Wenn Sie Ihr Programm in kleine Quelldateien unterteilen, es aber als eine Datei assemblieren wollen, kann die Hauptdatei aus INCLUDEs für alle die kleineren Quelldateien bestehen. Aber aus Gründen, die wir in Kapitel 5 besprochen haben, ist in den meisten Fällen das Assemblieren kleiner Module und ihr Linken mit LINK vermutlich der bessere Weg.

Eine andere Möglichkeit für eine INCLUDE-Datei ist eine Datenstruktur. Sie können eine bestimmte Datenstruktur in mehreren Programmen benutzen. Sie können die Definition dieser Struktur als eine getrennte Datei führen und mit der INCLUDE-Anweisung in jedem Programm, das dies erforderlich macht, auf sie zugreifen. Wir werden später in diesem Kapitel noch ausführen, auf welche Weise ein Programm Datenstrukturen benutzen kann.

Wenn die eingefügte Datei eine Makrodatei ist, braucht sie nicht jedesmal, wenn Sie das Programm assemblieren, in der Assemblerliste zu erscheinen. Sie können die IF1-Anweisung benutzen, um die Makros aus der Liste zu löschen, sie aber dennoch für die Code-Erstellung verfügbar halten.

Die Befehlsfolge

```
IF1
INCLUDE      87MAC.LIB
ENDIF
```

bezieht die Datei 87MAC.LIB während des ersten Assemblerdurchlaufs ein. In diesem ersten Durchlauf erweitert der Assembler alle Makros zu ihrer endgültigen Form. Da der Assembler die Makrodefinitionen während des zweiten Durchlaufs nicht mehr benötigt, sollte das Programm sie nur für den ersten Durchlauf einbeziehen. Das beschleunigt das Assemblieren, da die Makrodatei während des zweiten Durchlaufs nicht mehr gelesen wird. Der Assembler druckt die Makrodatei auch nicht aus, da er die Listendatei während des zweiten Durchlaufs erstellt. Abbildung 6.11 zeigt den Gebrauch von IF1...INCLUDE...ENDIF für die 8087-Makros. Diese Abbildung enthält beide, die Quell- und Listendateien für das Programm. Der Assembler behandelt die 8087-Instruktionen korrekt ohne Auflistung der Makros, die sie definieren.

```
        PAGE    ,132
        TITLE   Figure 6.11 8087 Macro INCLUDE

IF1
INCLUDE 87MAC.LIB
ENDIF

CODE    SEGMENT
        ASSUME  CS:CODE, DS:CODE

TWO_BYTE        DW      ?

        FENI
        FIDIVR  TWO_BYTE
        FLD

CODE    ENDS
        END
```

Figure 6.11(a) Source Instructions

```
The IBM Personal Computer MACRO Assembler 01-01-83           PAGE    1-1
Figure 6.11 8087 Macro INCLUDE

1                                             PAGE    ,132
2                                             TITLE   Figure 6.11 8087 Macro INCLUDE
3
4                                     ENDIF
5
6       0000                          CODE    SEGMENT
7                                             ASSUME  CS:CODE, DS:CODE
8
9       0000  ????                    TWO_BYTE        DW      ?
10
11                                            FENI
12      0002  9B DB E0            +           DB      09BH,0DBH,0E0H
13                                            FIDIVR  TWO_BYTE
14      0005  9B                  +                DB      09BH
15      0006  DE 3E 0000 R        +                ESC     037H,TWO_BYTE
16                                            FLD
17      000A  9B D9 C1            +                DB      09BH,0D9H,0C1H
18
19      000D                          CODE    ENDS
20                                            END
Open conditionals: 1
```

Figure 6.11(b) Assembly Listing

Abbildung 6.11 INCLUDE mit einer 8087-Makrodatei:
(a) Quellprogramm; (b) Assemblerliste

Segmente

Wir haben schon früher die SEGMENT-Anweisung betrachtet. Jetzt können wir uns damit etwas genauer beschäftigen und sehen, was sie sonst noch kann.

In den meisten Beispielprogrammen, die wir bis jetzt benutzt haben, hat es jeweils eine einzelne SEGMENT-Anweisung gegeben. Da sich der erzeugte Programmcode in irgendeinem Segment befinden muß, müssen wir ein solches benennen. Da der Assembler in der Lage sein muß, die Segmentadressierung durchzuführen, identifiziert die ASSUME-Anweisung im Programm das einzige Segment. In einem Fall wie diesem nutzen wir die Segmentierfähigkeiten des 8088 nicht voll aus, und in vielen Fällen ist das auch gar nicht notwendig. Wenn ein Programm und sein Datenbereich innerhalb desselben 64K-Adressbereiches residieren, brauchen die Segmentierfähigkeiten des Prozessors nicht in Anspruch genommen zu werden.

Es gibt Fälle, in denen ein Programm mehr als eine Segmentanweisung benutzen muß. Mehrere der DOS-Beispiele im Kapitel 5 haben solchen Gebrauch dargestellt. In diesen Beispielen definierte das Programm ein STACK-Segment. Der Segmentname war bedeutungslos, aber die Segmentklasse, wie in der SEGMENT-Anweisung angegeben, mußte STACK sein. Dies deshalb, weil eine .EXE-Datei einen reservierten Stapelbereich für die Ausführung des Programmes erfordert. Hätte das Programm kein STACK-Segment eingerichtet, würde der DOS-Lader den Stack an einer möglicherweise ungeeigneten Stelle initialisieren. In diesem Falle würde das Programm vermutlich nicht sehr gut laufen.

Ein anderer Gebrauch der SEGMENT-Anweisung dient dem Auffinden von Daten an irgendeinem bestimmten Platz in der Maschine. Wie wir bei DOS gesehen haben, ist es am besten, wenn das Programm segmentverschiebbar ist. Auf diese Weise brauchen wir uns nicht darum zu kümmern, wohin DOS das Programm lädt. Aber es gibt auch Fälle, bei denen die genaue Adresse für Code oder Daten wichtig ist. In diesen Fällen können wir die AT-Direktive der SEGMENT-Anweisung benutzen, um die Adresse festzulegen.

Um den Wert der AT-Direktive festzustellen, sehen wir uns ein Beispiel an. Dieses Beispiel benutzt das ROM BIOS des PC als Ausgangspunkt. Obgleich die Assemblersprache ein sehr leistungsfähiges Programmierwerkzeug ist, ist sie andererseits aber auch sehr schwierig zu handhaben, besonders bei großen Programmen. Sie wählten die Assemblersprache nur wegen ihrer Fähigkeiten, die sie für bestimmte Aufgaben besonders geeignet macht. Beim IBM PC ist die Assemblersprache die beste Sprache für die Aufgaben, die das ROM BIOS ausführt. Wir können diese Anwendungen mit der Anforderung charakterisieren, ein I/O-Gerät zu steuern, was in der Regel bitsignifikante Operationen notwendig macht. Die Fähigkeit, genau ermittelte Speicherstellen und I/O-Ports zu handhaben, gehört zu diesem Programmieren. Assemblersprache wird auch dort benutzt, wo kleinste Codelänge oder größtmöglichste Ausführungsgeschwindigkeit verlangt werden. All dies sind Eigenschaften des ROM BIOS.

Unser Beispiel benutzt einen Teil des ROM BIOS. In einem späteren Kapitel wollen wir behandeln, wie man vorgehen muß, um Teile des ROM BIOS auszutauschen. Für

diesen Fall sind wir jedoch am Zugriff auf die Datenstrukturen interessiert, die das ROM BIOS benutzt. Wenn Sie die Assemblerliste des ROM BIOS prüfen (welche im Anhang A des Technical Reference Manual für den IBM PC zu finden ist), werden Sie sehen, daß sich das DATA-Segment bei Segment 40H oder der absoluten Adresse 400H befindet. Das Programm in Abbildung 6.12 greift auf die Daten im ROM BIOS-Datenbereich für einen bestimmten Zweck zu. Dort ist eine Variable, KB_FLAG im DATA-Segment, die den aktuellen Zustand der Shift-Tasten anzeigt. Eine der oft erhobenen Beschwerden hinsichtlich der IBM-Tastatur ist, daß man nicht sagen kann, ob man im CAPS LOCK-Zustand ist oder nicht. Das Programm der Abbildung 6.12 liest das CAPS LOCK-Datenbit und zeigt es in der oberen rechten Ecke des Farbbildschirms an. Obgleich das Programm dies nicht vorsieht, wollen wir annehmen, daß, wenn Sie diesen Codeabschnitt tatsächlich benutzen, die obere rechte Ecke des Bildschirmes für diese Anzeige freigehalten wird.

Das DATA-Segment in Abbildung 6.12 zeigt, wie der Programmierer absolute Adressinformation in das Programm übertragen kann. Die DATA SEGMENT-Anweisung benutzt die AT-Direktive, um das Segment unbedingt bei Paragraph 40H zu plazieren. Gehen wir die ROM BIOS-Auflistung weiter durch, zeigt sich, daß die

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.12  Segment Location

 1                                    PAGE     ,132
 2                                    TITLE    Figure 6.12  Segment Location
 3
 4      0000                  DATA    SEGMENT AT 40H
 5      0017                          ORG      17H
 6      0017  ??              KB_FLAG          DB       ?
 7      = 0040                CAPS_STATE       EQU      40H
 8      0018                  DATA    ENDS
 9
10      0000                  VIDEO   SEGMENT AT 0B800H
11      009E                          ORG      158
12      009E  ??              INDICATOR        DB       ?
13      009F                  VIDEO   ENDS
14
15      0000                  CODE    SEGMENT
16                                    ASSUME   CS:CODE
17
18      0000                  CAPS    PROC     FAR
19      0000  1E              START:  PUSH     DS                      ; Return address
20      0001  B8 0000                 MOV      AX,0
21      0004  50                      PUSH     AX
22      0005  B8  ---- R              MOV      AX,DATA                 ; DATA segment address
23      0008  8E D8                   MOV      DS,AX
24                                    ASSUME   DS:DATA
25      000A  B8  ---- R              MOV      AX,VIDEO                ; VIDEO segment address
26      000D  8E C0                   MOV      ES,AX
27                                    ASSUME   ES:VIDEO
28      000F                  DISPLAY_CAPS:
29      000F  B0 18                   MOV      AL,18H                  ; Determine CAPS state
30      0011  F6 06 0017 R 40         TEST     KB_FLAG,CAPS_STATE      ; Up arrow = 18H
31      0016  75 02                   JNZ      CAPS_LOCK
32      0018  B0 19                   MOV      AL,19H                  ; Down arrow = 19H
33      001A                  CAPS_LOCK:
34      001A  26: A2 009E R           MOV      INDICATOR,AL            ; Store flag in upper right
35      001E  B4 06                   MOV      AH,6                    ; Direct Console I/O
36      0020  B2 FF                   MOV      DL,0FFH                 ; Keyboard input
37      0022  CD 21                   INT      21H
38      0024  3C 00                   CMP      AL,0                    ; Test for keyboard input
39      0026  74 E7                   JZ       DISPLAY_CAPS            ; Skip if no input
40      0028  3C 25                   CMP      AL,'%'                  ; Test for exit character
41      002A  74 08                   JE       RETURN
42      002C  B4 02                   MOV      AH,2                    ; Display output
43      002E  8A D0                   MOV      DL,AL                   ; Character to output
44      0030  CD 21                   INT      21H
45      0032  EB DB                   JMP      DISPLAY_CAPS            ; Do it again
46      0034                  RETURN:
47      0034  CB                      RET                              ; Return to DOS
48      0035                  CAPS    ENDP
49      0035                  CODE    ENDS
50                                    END      START
```

Abbildung 6.12 Anordnung von Segmenten

Variable KB_FLAG sich bei Offset 17H im DATA-Segment befindet. ORG 17H im Programm ermittelt den Offset dieser Variablen für die Assemblierung. Schließlich kommt noch EQU für CAPS_STATE direkt aus der ROM BIOS-Liste. Dieses Bit zeigt den aktuellen Stand des CAPS LOCK-Umschalters an.

Es gibt in Abbildung 6.12 eine weitere SEGMENT-Anweisung. Es ist das VIDEO-Segment, eingesetzt bei 0B800H. Dies ist die Segmentadresse des Bildschirmpuffers für die Farbkarte. Wir brauchen diese Adresse, um die Anzeige auf den Bildschirm auszugeben. Wenn wir das Zeichen in die obere rechte Ecke bringen wollen und davon ausgehen, daß es sich um einen Bildschirm mit 80 Zeichen handelt, ist der korrekte Offset für das Zeichen 158 (dezimal). Kapitel 8 beschreibt die Programmiereigenschaften der Hardware, so daß Sie dies zunächst einmal glauben müssen.

Der erste Teil des Programmes erstellt die erforderliche Segmentadressierung. Das DS-Register zeigt zum DATA-Segment, ES zum VIDEO-Segment. Obgleich das Programm die Segmente als absolut erklärt und dafür die AT-Deklaration benutzt, zeigt der Assembler sie weiterhin mit dem „R"-Attribut als verschieblich an. Das LINK-Programm setzt jedoch wiederum die korrekten Werte in diese Datenfelder ein.

Das Programm testet KB_FLAG und erlaubt dem Assembler, den korrekten Offset 17H zu erstellen. Das Beispiel benutzt das Symbol „↓", um den Normalfall anzuzeigen, das Symbol „↑", um CAPS LOCK anzugeben. Wir lesen die Tastatureingabe mit Hilfe der DOS-Funktion und stellen das Zeichen dar, soweit eines eingegeben wurde. Im Beispiel wurde das Symbol „%" willkürlich gewählt, um das Programm zu beenden. Gibt der Benutzer irgendein anderes Zeichen ein, zeigt das Programm dieses Zeichen auf dem Bildschirm und kehrt zur weiteren Eingabe zurück.

Sollten Sie dieses Programm eingeben und ausführen, sehen Sie den nach oben oder nach unten gerichteten Pfeil in der oberen rechten Ecke des Farbbildschirms dargestellt. Ist der Farbbildschirm bei Ausführung dieses Programmes im 40-Zeichen-Modus, erscheint die Pfeilanzeige in der zweiten Zeile von oben. Wenn Sie das Programm mit der Schwarz/Weiß-Karte laufen lassen wollen, verändern Sie das VIDEO-Segment nach 0B000H, der Adresse der Schwarz/Weiß-Karte.

Wenn Sie das Programm mit der Farbkarte im 80-Zeichen-Modus ausführen, werden Sie eine Menge „Schnee" auf dem Bildschirm sehen. Diese Bildstörung tritt auf, wenn das Programm direkt in den Bildschirmpuffer schreibt. Die Störung erscheint nicht bei der Schwarz/Weiß-Karte oder bei der Farbkarte im 40-Zeichen-Modus. Wir werden noch sehen, warum das so ist und wie man es umgehen kann, wenn wir die Hardware des IBM PC behandeln.

Es gibt noch weitere Anwendungen für mehrfache SEGMENT-Anweisungen in einem Programm. Wenn ein Programm einen größeren Datenbereich als 64K braucht, muß es den Zugriff auf diese Daten verwalten. Wie allgemein üblich, werden Sie den Datenbereich nach irgendeinem Speicherverwaltungsschema bearbeiten. In einem solchen Fall könnten Sie den gesamten Datenbereich (mit Ausnahme einiger fixierter Bereiche) über indirekte Bezugsadressen verwalten.

Als Beispiel wollen wir uns ansehen, wie der Kommandointerpreter für DOS Programme lädt. DOS lädt ein transientes Programm an der Paragraphengrenze, die dem residenten Teil von DOS folgt. Die Länge des residenten Teiles kann unterschiedlich sein, je nach Anzahl der Diskettenlaufwerke im System. Auch der Gebrauch der DOS-Unterbrechung INT 27H, welche das Programm ohne Speicherfreigabe beendet, verlängert DOS effektiv. Der DOS-Programmlader muß allerdings das Programmsegment-Präfix (PSP) des Programmes, das er lädt, adressieren. Der einfachste Weg, diese Datenstruktur zu definieren, ist eine getrennte SEGMENT-Anweisung.

Abbildung 6.13 zeigt eine SEGMENT-Deklaration, die an zwei verschiedenen Stellen benutzt werden kann. Könnten wir den Quellcode für den DOS-Lader sehen, könnten wir dort möglicherweise eine derartige Definition finden. Für ein Programm, das eine .EXE-Struktur benutzt, könnten wir eine solche Segmentstruktur haben, die den Zugriff auf die Variablen im PSP ermöglicht. In Abbildung 5.6, dem DOS-Beispiel, wurde eine .COM-Dateistruktur benutzt. Dies gestattete es uns, die verschiedenen Adressen im PSP mit einem Offset relativ zum PSP anzusprechen. DOS hatte den Programmcode in dasselbe Segment geladen, das auch das PSP enthielt, und das ganze damit sehr einfach gemacht.

Bei einer .EXE-Datei befindet sich das PSP in einem anderen Segment als der Code. Da DOS das DS- und ES-Register auf den Anfang des PSP setzt, wenn die Steue-

```
The IBM Personal Computer MACRO Assembler 01-01-83         PAGE    1-1
Figure 6.13 Program Segment Prefix

1                                                  PAGE    ,132
2                                                  TITLE   Figure 6.13 Program Segment Prefix
3       0000                               PROGRAM_SEGMENT_PREFIX  SEGMENT
4
5       0000       02 [                    INT_20          DB      2 DUP(?)
6                          ??
7                              ]
8
9       0002  ????                         MEMORY_SIZE     DW      ?
10      0004       05 [                    LONG_CALL       DB      5 DUP(?)
11                         ??
12                             ]
13
14      0009  ????????                     TERMINATE_ADDR  DD      ?
15      000D  ????????                     CTRL_BREAK      DD      ?
16      005C                                       ORG     05CH
17      005C       10 [                    FCB1            DB      16 DUP(?)
18                         ??
19                             ]
20
21      006C                                       ORG     06CH
22      006C       10 [                    FCB2            DB      16 DUP(?)
23                         ??
24                             ]
25
26      0080                                       ORG     080H
27      0080       80 [                    DTA             DB      128 DUP(?)
28                         ??
29                             ]
30
31
32      0100                               PROGRAM_SEGMENT_PREFIX  ENDS
33
34      0000                               CODE    SEGMENT
35                                                 ASSUME  CS:CODE,DS:PROGRAM_SEGMENT_PREFIX
36
37      0000  A1 0002 R                            MOV     AX,MEMORY_SIZE
38
39      0003                               CODE    ENDS
40                                                 END
```

Abbildung 6.13 Programmsegment-Präfix (PSP)

rung an das .EXE-Programm übergeht, ist es sinnvoll, das PSP als ein getrenntes Segment zu behandeln. Der restliche Code im CODE-Segment der Abbildung 6.13 zeigt, wie Sie auf die Daten im PSP zugreifen können.

Beachten Sie, daß das PSP-Segment in Abbildung 6.13 tatsächlich noch keine Werte für die Datenvariablen enthält. Zum Beispiel wissen wir, daß die ersten beiden Bytes des PSP den Code für INT 20H enthalten. Wir haben das Beispiel gewählt um zu zeigen, daß es an dieser Adresse ein 2-Byte Datenfeld ohne jeden Bezug zu den dort befindlichen Daten gibt. Dies ist notwendig, damit der Linker und der Lader nicht versuchen, als Ergebnis dieser Segmentdeklaration dort irgendwelche Daten abzuspeichern. Wir benutzen dieses Segment tatsächlich als Hilfsmittel zur Datendeklaration. Die Segmentanweisung definiert eine Datenstruktur, die wir Programmsegment-Präfix nennen. Es ist kein fester Bereich, kann aber über eines der Segmentregister adressiert werden. In unserem Beispiel in Abbildung 6.13 wird sie über das DS-Register angesprochen.

Die gleiche Methode können wir benutzen, um jede Art von Datenstruktur zu identifizieren, die sich beliebig im Speicherbereich des 8088 befinden kann. Die Datenstruktur könnte ein Steuerblock für das Betriebssystem sein; sie könnte eine einzelne Textzeile für den Text-Editor sein; oder sie könnte sogar der Parameterblock für ein bestimmtes Unterprogramm sein. In jedem dieser Beispiele residiert die Datenstruktur in ihrem eigenen Segment. Das heißt, daß ein Programm jedes Element der Datenstruktur mit dem Segmentregister anspricht, das auf den Beginn (oder in die Nähe des Beginns) des Elementes weist. Das Programm greift mit demselben Segmentregisterwert nicht auf zwei verschiedene Elemente zu. Es setzt für jedes Element den Segmentwert neu.

An dieser Stelle sollten wir uns kurz über Strategien zur Speicherzuteilung beim 8088 unterhalten. Der IBM PC mit dem 8088 kann bis zu 1 Megabyte Speicherplätze adressieren, aber ein einzelnes Segment kann nur 64K Bytes enthalten. Selbst mit den vier Segmentregistern gibt es keine Möglichkeit für ein Programm, alle Stellen des Speichers ohne einige Segmentierstrategien zu erreichen.

Wenn die Daten alle innerhalb von 64K Bytes untergebracht werden können, ist weiter nichts zu unternehmen. Sie setzen lediglich alle Daten in das gleiche Segment. Nehmen wir aber an, daß ein Programm einen Datenbereich braucht, der 64K Bytes übersteigt, so müssen wir das Problem der Speicherverwaltung lösen. Sie können dazu zwei Strategien verwenden. In beiden Fällen wollen wir annehmen, daß die Daten in kleinere Einheiten unterteilt werden können (wie z. B. einzelne Variablen, Textzeilen, Steuerblöcke oder Datenfelder), von denen jede kleiner als 64K Bytes ist.

Sie sollten die erste Zuordnungsmethode benutzen, wenn Ihre vorrangige Absicht in der Erhaltung von Speicherplatz besteht. Mit dieser Methode ermitteln Sie Datenobjekte an der ersten freien Speicherstelle. Dies erfordert, daß das Programm, das die Datenbereiche verwaltet, einen 4-Byte Zeiger für jede Datenvariable benutzt. Zwei Bytes sind für den Offset und zwei weitere für die Segmentadresse bestimmt.

Wenn das Programm auf die Daten zugreifen will, nimmt es die Adresse der Daten aus dem Adresspeicherbereich mit einer LDS- oder LES-Anweisung wieder auf.

Wollen Sie noch mehr Platz sparen, können Sie die Adresszeiger sogar in einem Drei-Byte-Feld ablegen. Zwei Bytes enthalten dann die Segmentadresse der Daten. Das verbleibende Byte ist der Offset des Objektes in diesem Segment. Der Start-Offset sollte immer ein Wert zwischen 0 und 15 sein, da die Segmentadresse ein Vielfaches von 16 sein kann.

Obwohl diese Methode äußerst platzsparend für die Speicherung von Daten ist, hat sie auch einige Haken. Die maximale Größe eines Datenobjektes ist etwas kleiner als 64K Bytes. Im ungünstigsten Falle endet bei dieser Zuordnungsstrategie die absolute Adresse des Datenobjekts mit 0FH, was einen Start-Offset von 0FH erfordert. Da der maximale Offset in jedem Segment 0FFFFH ist, ist die maximale Länge der Variablen 64K-15, oder 65,521 Bytes. Ein zweiter Nachteil bei dieser Methode ist der Speicherplatz, der für die Sicherung der Adresspointer auf die Datenobjekte erforderlich ist. Wenn es sich um eine große Anzahl von Objekten handelt, nimmt die Summe aller 4-Byte (oder 3-Byte) Zeiger einen großen Speicherraum ein.

Ein Beispiel für diese Methode der Speicherzuordnung ist der Dateisteuerblock FCB (File Control Block). In dem vorangegangenen DOS-Beispiel haben wir dem FCB einen willkürlichen Ort im Programm zugeordnet. Es gab keine bestimmte Ausrichtung dieser Datenstruktur. Als wir dann DOS aufriefen, um eine Dateioperation durchzuführen, benötigte das Programm einen 4-Byte Pointer. Das DS:DX-Registerpaar identifizierte den FCB für DOS.

Die zweite Methode der Speicherzuordnung setzt alle Datenobjekte an eine Paragraphengrenze. Das vereinfacht sofort die Adresspointer, die das Datenobjekt identifizieren. Der Pointer besteht nur noch aus zwei Bytes. Dieser 2-Byte Wert bestimmt das Segment, in welchem die Daten residieren. Da die Datenbereiche immer an einer Paragraphengrenze liegen, ist der Offset für die Daten immer Null. Diese Methode benötigt jedoch sehr viel Speicherplatz. Jedesmal, wenn Sie Speicher für ein neues Objekt zuordnen, werden möglicherweise 15 Bytes verschwendet. Dies trifft dann zu, wenn das letzte Byte des vorhergehenden Objektes direkt auf einer Paragraphengrenze liegt. Die nächste Paragraphengrenze liegt 15 Bytes weiter, und die 15 dazwischenliegenden Bytes werden vergeudet. Außerdem beträgt die minimale Größe für ein Objekt 16 Bytes. Ist irgendein Datenbereich kleiner, dann sind die restlichen Bytes ohnehin verloren.

Der DOS-Lader benutzt, wie wir es besprochen haben, die zweite Methode der Speicherzuordnung, wenn er Programme lädt. DOS lädt ein Programm immer an der nächsten Paragraphengrenze. Da DOS davon ausgeht, daß eine kleine Anzahl von großen Objekten im Speicher residiert, ist diese Methode in Bezug auf den Speicherplatz nicht sehr verschwenderisch. Wenn Ihre Anwendung jedoch viele kleine Objekte erfordert, kann die Paragraphenausrichtung möglicherweise zu aufwendig werden.

Die zweite Zuordnungsmethode, die die Paragraphenausrichtung benutzt, läßt Sie den Datenbereich mittels SEGMENT-Datenstrukturen definieren. Wenn Sie die erste Methode benutzen wollen, müssen Sie einen anderen Weg zur Definition der Datenstruktur gehen. Der nächste Abschnitt behandelt ein Werkzeug für eine solche Datendeklaration.

Strukturen

Eine Datenstruktur ist eine Datenanordnung, die für den Programmierer bedeutungsvoll ist. In der Regel definieren wir Datenstrukturen dann, wenn mehr als ein Programm oder Programmierer denselben Datensatz benutzen. Durch das Definieren der Datenstruktur haben beide Parteien eine klare Vorstellung von den Daten. Wenn Programm A Daten an Programm B weitergibt, stellt die definierte Datenstruktur sicher, daß jedes Programm an der gleichen Adresse nach den Daten sucht.

Wir haben bereits ein gutes Beispiel einer Datenstruktur. Der Dateisteuerblock (FCB) ist eine Datenstruktur. Programme benutzen den FCB, um Dateiinformation an DOS zu übermitteln. Der FCB enthält wichtige Daten über die offene Datei — Werte wie die aktuelle Datensatznummer, Dateigröße usw. Der FCB enthält außerdem Daten, die nur von DOS benutzt werden — das reservierte Feld. Alle Daten, die für DOS und das Anwendungsprogramm erforderlich sind, befinden sich im FCB. Diese Datenstruktur übermittelt also die Dateiparameter zwischen DOS und dem Anwendungsprogramm.

Wir brauchen nun einen Weg, um Datenstrukturen so zu definieren, daß ein Programm sie in günstiger Form benutzen kann. Der IBM Makro-Assembler verfügt dazu über die Anweisung STRUC, die Sie eine Datenstruktur definieren läßt. Aus der Sicht des Programmierers scheint die Datenstruktur nur ein weiteres Segment zu sein. Die Datendefinition wird genauso wie normale Datenanweisungen assembliert, und die Struktur mit einer ENDS-Anweisung wie ein Segment beendet. Es werden jedoch keine Daten im eigentlichen Sinn erzeugt. Der STRUC-Befehl übermittelt dem Assembler nur die Struktur der Daten. In einem späteren Teil der Assemblierung wird dann der Name der Struktur verwendet, um den Datenbereich zu erstellen.

Aus dieser Perspektive erscheint die STRUC-Anweisung mehr wie ein MACRO. Ein Programm definiert die Struktur in einem Abschnitt und ruft die Struktur zu einem späteren Zeitpunkt auf. Beim Aufruf der Struktur werden die Daten tatsächlich erstellt. Abbildung 6.14 wird uns helfen, die Operation der STRUC-Anweisung zu verstehen.

Abbildung 6.14 enthält ein sehr einfaches Programm, das DOS-Dateien benutzt. Dieses Programm eröffnet eine DOS-Datei im Laufwerk A:, liest den zweiten Satz dieser Datei und schreibt diesen Datensatz in eine DOS-Datei im Laufwerk B:. Es ist unwahrscheinlich, daß Sie ein solches Programm jemals für einen wichtigen Vorgang schreiben werden, aber es gibt uns die Möglichkeit, die Datenstruktur für den FCB zu benutzen.

Der erste Teil des Programms in Abbildung 6.14 definiert die Datenstruktur FCB. Die Assembleranweisung STRUC kennzeichnet den Beginn der Strukturdefinition. Das Label FCB benennt diese spezielle Struktur. Das Beispiel definiert jedes der Felder des FCB in der Struktur. Beachten Sie, daß der Assembler den Assemblercode für die Struktur in den linken Spalten erstellt. Wenn jedoch der erzeugte Objektcode gelinkt ist, gibt es keinen Datenbereich im Programm. Der Assembler hat eine assemblierte Version der Datenstruktur lediglich für Ihre Referenzzwecke miteinbezogen.

```
The IBM Personal Computer MACRO Assembler 01-01-83           PAGE    1-1
Figure 6.14  Structures

1                                               PAGE    ,132
2                                               TITLE   Figure 6.14  Structures
3
4                                       FCB     STRUC
5        0000  00                       DRIVE           DB      0               ; Drive Number
6        0001  20 20 20 20 20 20        FILE_NAME       DB      '        '      ; File name
7              20 20
8        0009  20 20 20                 FILE_EXT        DB      '   '           ; Extension
9        000C  0000                     CURRENT_BLOCK   DW      0               ; Current block number
10       000E  0080                     RECORD_SIZE     DW      80H             ; Logical record size
11       0010  00 00 00 00              FILE_SIZE       DD      0               ; Size of the file, in bytes
12       0014  0000                     DATE            DW      0               ; Date the file was created
13       0016     0A [                  RESERVED        DB      10 DUP(?)       ; Reserved, can't be overriden
14                        ??
15                            ]
16
17       0020  00                       SEQ_NUMBER      DB      0               ; Record in block for sequential
18       0021  00 00 00 00              RANDOM_NUMBER   DD      0               ; Record in file for random
19       0025                           FCB     ENDS
20
21       0000                           STACK   SEGMENT STACK
22       0000     40 [                          DW      64 DUP(?)
23                        ????
24                              ]
25
26       0080                           STACK   ENDS
27
28       0000                           CODE    SEGMENT
29                                              ASSUME  CS:CODE
30
31       0000  01                       INPUT   FCB     <1,'FIG6-14','INP'>
32       0001  46 49 47 36 2D 31
33             34 20
34       0009  49 4E 50
35       000C  0000
36       000E  0080
37       0010  00 00 00 00
38       0014  0000
39       0016     0A [
40                        ??
41                            ]
42       0020  00
43       0021  00 00 00 00
44
45
46       0025  02                       OUTPUT  FCB     <2,'EXAMPLE','TST'>
47       0026  45 58 41 4D 50 4C
48             45 20
49       002E  54 53 54
50       0031  0000
51       0033  0080
52       0035  00 00 00 00
53       0039  0000
54       003B     0A [
55                        ??
56                            ]
57       0045  00
58       0046  00 00 00 00
59
60
61       004A                           STRUCTURES      PROC    FAR
62       004A  1E                               PUSH    DS                      ; Set return address
63       004B  B8 0000                          MOV     AX,0
64       004E  50                               PUSH    AX
65       004F  0E                               PUSH    CS                      ; Set DS into CODE segment
66       0050  1F                               POP     DS
67                                              ASSUME  DS:CODE
68       0051  8D 16 0000 R                     LEA     DX,INPUT                ; Open the input file
69       0055  B4 0F                            MOV     AH,0FH
70       0057  CD 21                            INT     21H
71
72       0059  8D 16 0025 R                     LEA     DX,OUTPUT               ; Create the output file
73       005D  B4 16                            MOV     AH,16H
74       005F  CD 21                            INT     21H
75
76       0061  8D 1E 0000 R                     LEA     BX,INPUT
77       0065  C7 47 0E 0010                    MOV     [BX].RECORD_SIZE,16     ; Set record size for input file
78       006A  C6 47 20 01                      MOV     [BX].SEQ_NUMBER,1       ; Skip first record
79
80       006E  C7 06 0033 R 0010                MOV     OUTPUT.RECORD_SIZE,16   ; Set record size for output
81
82       0074  8D 16 0000 R                     LEA     DX,INPUT                ; Read second record in file
83       0078  B4 14                            MOV     AH,14H
84       007A  CD 21                            INT     21H
85
86       007C  8D 16 0025 R                     LEA     DX,OUTPUT               ; Write that record to ouput
87       0080  B4 15                            MOV     AH,15H
88       0082  CD 21                            INT     21H
89
```

```
90        0084  B4 10                 MOV     AH,10H              ; Close the output file
91        0086  CD 21                 INT     21H
92
93        0088  CB                    RET
94        0089                STRUCTURES      ENDP
95        0089                CODE    ENDS
96                                    END     STRUCTURES

The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    Symbols-1
Figure 6.14  Structures

Structures and records:

                N a m e               Width   # fields
                                      Shift   Width   Mask    Initial
FCB. . . . . . . . . . . . . .        0025    000A
  DRIVE. . . . . . . . . . . . . .    0000
  FILE_NAME. . . . . . . . . . . .    0001
  FILE_EXT . . . . . . . . . . . .    0009
  CURRENT_BLOCK. . . . . . . . . .    000C
  RECORD_SIZE. . . . . . . . . . .    000E
  FILE_SIZE. . . . . . . . . . . .    0010
  DATE . . . . . . . . . . . . . .    0014
  RESERVED . . . . . . . . . . . .    0016
  SEQ_NUMBER . . . . . . . . . . .    0020
  RANDOM_NUMBER. . . . . . . . . .    0021
```

Abbildung 6.14 Strukturen

Der Name FCB ist nun zu einem neuen Assemblerbefehl geworden, so als wäre er ein Makro. Die erste Anweisung im CODE-Segment ist ein Aufruf der FCB-Datenstruktur. Das Beispiel gibt der Struktur den Namen INPUT. Dieser FCB identifiziert den Eingabedatensatz. Beachten Sie, daß es im Zusammenhang mit der FCB-Anweisung Operanden gibt. Diese Operanden ersetzen oder überschreiben die in der ursprünglichen Definition der Datenstruktur enthaltenen Werte.

Wenn wir den Objektcode für die Definition der FCB-Struktur mit dem Objektcode für die INPUT-Struktur vergleichen, sehen wir, daß er sich in den ersten drei Feldern der Struktur unterscheidet. Die Definition enthält 0 im DRIVE-Feld, während bei INPUT dort 1 steht. Der erste Operand in den spitzen Klammern der INPUT-Definition ist eine 1. Dieser Wert überschreibt die ursprünglich definierte 0. In ähnlicher Weise überschreibt das Beispiel die zweiten und dritten Felder, die die Datei benennen. Die abschließende spitze Klammer in der INPUT-Definition beendet die Austauschwerte für die Strukturfelder. Der Rest der INPUT-Struktur ist identisch mit der FCB-Definition.

Ein Programm kann jedes Feld der FCB-Struktur ändern, wenn die ursprüngliche Definition dieses Feldes nur aus einem einzigen Datenbereich besteht. In unserem Beispiel kann das Programm jedes Feld im FCB modifizieren, mit Ausnahme des RESERVED-Feldes. Wir haben dieses Feld aus zehn Datenbereichen bestehend definiert, und das kann nicht geändert werden. In ähnlicher Weise können Sie ein Feld, das als

```
DB      10,20
```

definiert ist, nicht überschreiben. Nur Felder mit einem einzigen Datenwert im STRUC-Aufruf können überschrieben werden. Der Assembler behandelt auch einen Zeichenstring, der aus mehreren Zeichen zusammengesetzt ist, als einen Datenwert. In unserem Beispiel besteht das Feld FILE_NAME aus vielenZeichen, ist aber ein einziger Eintrag, den Sie folglich modifizieren können.

Die Operanden in den spitzen Klammern ersetzen die definierten Operanden entsprechend ihrer Position genau wie in einem Makro. Wollen Sie einen definierten Wert nicht modifizieren, aber das nächste Feld ändern, müssen Sie einen 0-Parameter in die Parameterliste einsetzen. Um zum Beispiel FILE_NAME und CURRENT_ BLOCK zu modifizieren, während der voreingestellte Wert für DRIVE und FILE_EXT erhalten bleibt, würden Sie die FCB-Struktur mit folgender Anweisung aufrufen:

EXAMPLE FCB <,NEWNAME,,12>

Das erste Parameterfeld ist Null, deshalb benutzt der Assembler die Voreinstellung. NEWNAME ersetzt den leeren String im nächsten Feld. Der voreingestellte Leerstring wird für FILE_EXT benutzt, und schließlich ersetzt 12 die 0 als Wert in CURRENT_BLOCK.

Die nächste Quellzeile im Programm der Abbildung 6.14 definiert den FCB für die Ausgabedatei mit dem Namen OUTPUT. Hier modifiziert das Beispiel wieder die ersten drei Felder der Datendefinition, indem es die neuen Werte im Operandenfeld der FCB-Anweisung verwendet.

Der Wert des Gebrauchs einer Datenstruktur zeigt sich in den tatsächlichen Befehlen für das Programm. Das Programm kann INPUT und OUTPUT wie alle anderen Labels ansprechen. Sie sehen dies in dem Codeabschnitt, der die INPUT-Datei eröffnet. Mit der Anweisung

LEA DX,INPUT

wird die Adresse des Eingabe-FCB erstellt.

Wir können jedes Feld der Datenstruktur im Programm benutzen. Der Wert eines Symbols ist der Offset dieses Feldes in der Datenstruktur. Zum Beispiel setzt das Programm das BX-Register mit der Adresse des INPUT FCB. Wir greifen dann über Basisregister-Adressierung auf die Felder RECORD_SIZE und SEQ_NUMBER zu. Da BX bereits auf den FCB zeigt, müssen wir den Offset von dieser Basis spezifizieren. Der Adressausdruck

[BX].RECORD_SIZE

weist den Assembler an, einen Befehl zu erstellen, der den Offset von RECORD_ SIZE in der Datenstruktur auf den Basiswert im BX-Register addiert. Wenn Sie den Maschinencode für die Anweisungen prüfen, sehen Sie, daß der Offset für RECORD_SIZE (0EH) und SEQ_NUMBER (20H) in den Anweisungen erscheint. Das „."-Symbol identifiziert die Feldernamen als Offset in einer Datenstruktur.

Außer für relative und indizierte Adressierung können Sie die Datenstruktur für direktes Adressieren verwenden. Der nächste Teil des Programmes modifiziert das Feld RECORD_SIZE des OUTPUT FCB direkt. Das Programm benennt dieses Feld als OUTPUT.RECORD_SIZE. Das Symbol OUTPUT identifiziert die spezielle Datenstruktur. RECORD_SIZE benennt das Feld in dieser Datenstruktur.

Bevor wir dieses Beispiel hinter uns lassen, wollen wir uns einmal ansehen, was der Assembler über die Datenstruktur weiß. Die Abbildung beinhaltet einen Abschnitt der Symboltabelle. Der Assembler reserviert einen Teil der Symboltabelle für Strukturen und Datensätze. Der Assembler zeigt Ihnen die Informationen, die er über die

Struktur besitzt. Der Abschnitt hat den Titel „Strukturen und Datensätze". Für die FCB-Struktur im Beispiel zeigt die erste Zeile, daß die Struktur eine Länge von 25H Bytes hat und 0AH Felder enthält. Der Assembler zeigt darüber hinaus jedes der unter dem Strukturnamen ansprechbaren Felder. Die Symboltabelle enthält den Offset jedes dieser Felder. Für Strukturen verwendet der Assembler die beiden als „width" und „# fields" gekennzeichneten Spalten. Die zweite dieser Spalten wird für Records benutzt. Im nächsten Abschnitt wollen wir diese Datenstrukturen behandeln, die dem Assembler als Records bekannt sind.

Das Programmbeispiel in Abbildung 6.14 erfüllt keinerlei nützlichen Zweck. Es enthält auch keinerlei Fehlerbearbeitung. Aber es illustriert den Gebrauch der STRUC-Anweisung. Dieses Werkzeug für die Datendefinition ist besonders nützlich bei oft gebrauchten Datenstrukturen. Der Gebrauch der Feldernamen für Offsetwerte ist sehr hilfreich, wenn Sie die Datenstrukturen während der Programmentwicklung modifizieren. Wenn Sie die Datenstruktur ändern, ändert der Assembler automatisch die Offsetwerte, wenn das Programm wieder assembliert wird. Außerdem trägt der Gebrauch von Datenstrukturen dazu bei, ein Assemblerprogramm besser lesbar und verständlich zu machen.

Records

Die Strukturen des vorhergehenden Abschnittes sind für Strukturen mit vielen Bytes bestimmt. Es gibt aber Situationen, in denen Sie Datenobjekte Bit für Bit identifizieren müssen. Für diesen Fall bietet der Makro-Assembler einen Datendefinitionsmechanismus mit dem Namen RECORD.

RECORD arbeitet ähnlich wie STRUC und MAKRO. Die RECORD-Anweisung definiert die Datenstruktur. Der mit RECORD verknüpfte Name wird ein weiterer Operator für den Assembler. Sie können Namen benutzen, um spezifische Daten zu definieren. RECORD unterscheidet sich von STRUC dadurch, daß es Objekte auf Bitebene definiert. Die RECORD-Anweisung gibt jedem der Felder einen Namen, zusammen mit der Breite des Feldes in Bits. Sie können die RECORD-Anweisung benutzen, um Bitfelder bis zu einer Gesamtlänge von 16 Bits zu erstellen.

Auch hier wollen wir wieder mit einem Beispiel arbeiten. Abbildung 6.15 zeigt ein weiteres, für nichts brauchbares Programm. Es befaßt sich mit dem Auffinden des Datums einer Datei. Die Definition des Dateisteuerblocks enthält ein 16-Bit Feld, welches das Datum beinhaltet, an dem DOS die Datei erstellt oder zum letzten Mal aktualisiert hat. Wenn DOS eine Datei eröffnet, füllt es dieses Feld im FCB aus der Datumsinformation, die im Disketteninhaltsverzeichnis gespeichert ist, auf. Das Datumsfeld enthält Jahr, Monat und Tag, kodiert in 16 Bits. Die RECORD-Anweisung der Abbildung 6.15 zeigt das Layout dieses Wortes.

In Abbildung 6.15 gibt das Beispiel dem Record den Namen DATE_WORD. Das RECORD-Operandenfeld zeigt, daß es in DATE_WORD drei Felder gibt. Die ersten 7 Bits sind das Jahr (YEAR), die nächsten 4 der Monat (MONTH), und die letzten 5 Bits enthalten den Tag (DAY). Genau wie bei einem MACRO definiert dieser Befehl lediglich den speziellen Datensatztyp, hier genannt DATE_WORD. Die RECORD-Anweisung erstellt die gespeicherten Daten nicht, bis Sie den RECORD-Namen als einen Assembleroperator benutzen.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 6.15  Records

1                                            PAGE    ,132
2                                            TITLE   Figure 6.15  Records
3
4                                    DATE_WORD       RECORD  YEAR:7,MONTH:4,DAY:5
5
6       0000                         STACK   SEGMENT STACK
7       0000      40 [                       DW      64 DUP(?)
8                       ????
9                             ]
10
11      0080                         STACK   ENDS
12
13      0000                         CODE    SEGMENT
14                                           ASSUME  CS:CODE
15
16      0000                         FCB     LABEL   BYTE
17      0000  01                     DRIVE           DB      1              ; Drive Number
18      0001  46 49 47 36 2D 31      FILE_NAME       DB      'FIG6-15 '     ; File name
19            35 20
20      0009  41 53 4D               FILE_EXT        DB      'ASM'          ; Extension
21      000C  0000                   CURRENT_BLOCK   DW      0              ; Current block number
22      000E  0080                   RECORD_SIZE     DW      80H            ; Logical record size
23      0010  00 00 00 00            FILE_SIZE       DD      0              ; Size of the file, in bytes
24      0014  00 00                  DATE            DATE_WORD     <>       ; Date the file was created
25      0016     0A [                RESERVED        DB      10 DUP(?)      ; Reserved, can't be overriden
26                      ??
27                          ]
28
29      0020  00                     SEQ_NUMBER      DB      0              ; Record in block for sequential
30      0021  00 00 00 00            RANDOM_NUMBER   DD      0              ; Record in file for random
31
32      0025                         RECORDS         PROC    FAR
33      0025  1E                             PUSH    DS                     ; Set return address
34      0026  B8 0000                        MOV     AX,0
35      0029  50                             PUSH    AX
36      002A  0E                             PUSH    CS                     ; Set DS into CODE segment
37      002B  1F                             POP     DS
38                                           ASSUME  DS:CODE
39      002C  8D 16 0000 R                   LEA     DX,FCB                 ; Open the file
40      0030  B4 0F                          MOV     AH,0FH
41      0032  CD 21                          INT     21H
42
43      0034  A1 0014 R                      MOV     AX,DATE
44      0037  25 FE00                        AND     AX,MASK YEAR           ; Isolate the year
45      003A  B1 09                          MOV     CL,YEAR                ; Right justify the value
46      003C  D3 E8                          SHR     AX,CL
47      003E  8A F8                          MOV     BH,AL                  ; Store year in BH
48
49      0040  A1 0014 R                      MOV     AX,DATE
50      0043  25 01E0                        AND     AX,MASK MONTH
51      0046  B1 05                          MOV     CL,MONTH
52      0048  D3 E8                          SHR     AX,CL
53      004A  8A D8                          MOV     BL,AL                 ; Store month in BL
54
55      004C  A1 0014 R                      MOV     AX,DATE
56      004F  25 001F                        AND     AX,MASK DAY           ; Store day in AL
57
58      0052  CB                             RET
59      0053                         RECORDS         ENDP
60      0053                         CODE    ENDS
61                                           END     RECORDS

The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    Symbols-1
Figure 6.15  Records

Structures and records:

                N a m e              Width   # fields
                                     Shift   Width   Mask    Initial

DATE_WORD. . . . . . . . . . . .     0010    0003
  YEAR . . . . . . . . . . . . .     0009    0007    FE00    0000
  MONTH. . . . . . . . . . . . .     0005    0004    01E0    0000
  DAY. . . . . . . . . . . . . .     0000    0005    001F    0000
```

Abbildung 6.15 Records

Das Beispiel erstellt DATE, indem es die DATE_WORD-Definition im FCB benutzt. Das Label DATE identifiziert diesen Datenbereich und DATE_WORD erstellt den 16-Bit Datenbereich. Wie wir später sehen werden, gibt es Methoden, die voreingestellten und die aktuellen Werte für die Felder in einem Datensatz zu definieren. In diesem Beispiel haben wir nichts unternommen, um den Standardwert von 0 für jedes der Felder zu modifizieren.

Über die Darstellung der Struktur der RECORD-Anweisung hinaus zeigt Ihnen das Programm in Abbildung 6.15 einige Operationen, die durch die RECORD-Anweisung vereinfacht werden. Der erste Teil des Programms eröffnet die im FCB benannte Datei. Das restliche Programm nimmt die Datumsinformation vom FCB und überträgt die einzelnen Felder in die Register des 8088.

Als erstes isoliert das Programm den Wert für das Jahr aus den Daten im FCB. Nachdem es DATE_WORD nach AX übertragen hat, setzt es die Werte für Monat und Tag in dem Wort mit einer AND-Anweisung auf 0. Beachten Sie hier den direkten Operanden MASK YEAR. Da YEAR ein Feld innerhalb eines Record ist, gibt der MASK-Operator einen Wert zurück, der YEAR innerhalb des Wortes isoliert. In diesem Falle gibt MASK YEAR den Wert 0FE00H zurück. Wie Sie daraus ersehen, sind die ersten 7 Bits 1, der Rest ist 0. Dieser Maskenwert entspricht den Bits, aus denen YEAR im Wort gebildet ist. Der AND-Befehl mit diesem Wert auf den Rest des Records läßt lediglich das YEAR-Feld übrig.

Mit den nächsten Instruktionen verschiebt das Programm das YEAR-Feld im Wort ganz nach rechts. Der Feldname YEAR hat einen Wert, der dem Shiftzähler entspricht, der für die Verschiebung des Feldes im Wort nach rechts erforderlich ist. In diesem Fall ist der Wert 9. Durch die Rechtsverschiebung mit 9 verbleibt das Jahr als eine Zahl in AL. (Beachten Sie, daß DOS das Jahr als eine Zahl zwischen 0 und 119 kodiert. Diese Werte entsprechen den Jahren 1980 bis 2099.)

Mit den nächsten Befehlen isoliert das Beispiel das MONTH-Feld aus dem Datensatz. Auch hier verwendet das Programm die MASK- und Verschiebewerte aus dem Record DATE_WORD. Und in ähnlicher Weise isoliert es auch das DAY-Feld.

Das Programm leistet keinerlei sinnvolle Arbeit, da die in die Register eingebrachten Werte verloren sind, sobald das Programm an DOS zurückkehrt. Sie können dieses Programm jedoch unter DEBUG benutzen und bei der Rückkehranweisung einen Breakpoint setzen. DEBUG zeigt das BH-, BL- und AL-Register auf dem Bildschirm, so daß Sie das Datum sehen können. Ein etwas nützlicheres Programm würde die Datumswerte aufgreifen und sie für die Bildschirmdarstellung in ASCII umwandeln. Oder Sie könnten dieses Programm als ein Unterprogramm umschreiben, das die Datumsinformation für ein anderes Programm liefert.

Es gibt noch einige andere Eigenschaften der RECORD-Operation, die wir behandeln sollten. Abbildung 6.15 enthält einen Teil der Symboltabelle aus der Assemblierung. Diese Tabelle zeigt die Information, die der Assembler über jedes der Felder im Record besitzt. Aus dieser Symboltabelle wollen wir uns die zweite Spalte „Shift Width Mask Initial" ansehen. Wie die Symboltabelle zeigt, ist das Record DATE_WORD 16 Bits breit und hat drei Felder. Jedes der Felder hat vier Attribute. Der Shift-Wert ist die Anzahl der Bits im Record rechts vom Feld. Dieser Wert gibt dem Assembler an, wie weit das Feld verschoben werden muß, um es ganz nach rechts auszurichten. Der Maskenwert isoliert das Feld. Eine 1 im Maskenfeld zeigt an, daß diese Position Teil des Feldes ist.

Die Feldbreite steht im Assembler für jedes Feld eines Record zur Verfügung. Mit dem Operator WIDTH können Sie zum Zeitpunkt der Assemblierung die Breite eines Feldes ermitteln.

In unserem Beispiel würde

MOV AL, WIDTH YEAR

den Wert 7 im AL-Register ablegen.

Die erste Spalte der Symboltabelle zeigt Ihnen an, welche Werte der Assembler bei Erstellung des Record einsetzt. Sie können Records mit Anfangswerten ungleich 0 spezifizieren. Sie können diese Werte dann zur Zeit der Generierung der Daten auch überschreiben. Um Anfangswerte zu definieren, setzen Sie hinter jede Feldbestimmung in der RECORD-Anweisung ein „=" und den jeweiligen Wert. Das Record DATE_WORD mit den Anfangswerten 1.Januar 1983 würde lauten:

DATE_WORD RECORD YEAR:7=3,MONTH:4=1,DAY:5=1

Sie können diese Werte in derselben Weise überschreiben, die Sie für Strukturen angewendet haben. Wenn Sie das Record erstellen, enthalten die spitzen Klammern die aktuellen Werte für die Generierung. Soll das Datum der 5.Januar 1984 sein, könnten Sie das folgende Record erstellen:

DATE DATE_WORD <4,,5>

Genau wie bei Makros und Strukturen sind die Argumente positionsabhängig. Da der Wert für den Monat bei 0 belassen wird, benutzt der Assembler den in der RECORD-Anweisung spezifizierten Anfangswert.

Beachten Sie, daß das Programm in Abbildung 6.15 die FCB-Struktur neu definiert, die STRUC-Anweisung des vorhergehenden Abschnittes aber nicht benutzt. Wir konnten STRUC nicht benutzen, weil jedes Feld in einer Struktur aus einer Datendefinition bestehen muß. Wir können aber keinen RECORD-Namen als Feld in einer Struktur benutzen. Der Weg, den wir in Abbildung 6.15 beschritten haben, ist eine Methode, dieses Problem zu umgehen.

Wir können dieses Problem auch auf andere Weise lösen. Auch wenn der Assembler keinen Datenbereich erstellt, bevor nicht der Name für das Record als Operator benutzt wird, speichert er doch die Felddefinitionen, wenn RECORD definiert wird. Dies ermöglicht es einem Programm, das Record DATE_WORD zu definieren, ohne es für die Spezifizierung des DATE-Feldes in der Struktur zu benutzen. Dies ist dasselbe, als wenn ein Makro definiert, aber niemals aufgerufen wird. Der Rest des Programms bleibt gleich. Die verschiedenen Feldernamen im Record DATE_WORD haben Bedeutung und können als Shiftzähler und MASK-Werte benutzt werden.

Dasselbe gilt für STRUC. Die Definition der Struktur definiert den Offset für den Assembler, auch wenn Sie den Strukturnamen niemals verwenden, um einen Datenbereich zu definieren. Sie können dies benutzen, um den voreingestellten FCB am Ort 05CH des Programm-Segment-Präfixes zu ermitteln. Dieser FCB ist immer vorhanden, so daß es nicht notwendig ist, die Struktur für die Generierung des Datenbereiches zu benutzen. Von mehreren Einzelheiten abgesehen, ist das Programm in Abbildung 6.16 mit der Abbildung 6.15 fast identisch. Das Program definiert einen FCB mit der STRUC-Anweisung anstelle einer Sammlung von DEFINE-Anweisungen. Beachten Sie, daß das Beispiel weder das Record

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE     1-1
Figure 6.16  Structure and Record

1                                              PAGE     ,132
2                                              TITLE    Figure 6.16  Structure and Record
3
4                                     DATE_WORD        RECORD  YEAR:7,MONTH:4,DAY:5
5
6                                     FCB     STRUC
7       0000  00                      DRIVE            DB      0                  ; Drive Number
8       0001  20 20 20 20 20 20       FILE_NAME        DB      '        '         ; File name
9             20 20
10      0009  20 20 20                FILE_EXT         DB      '   '              ; Extension
11      000C  0000                    CURRENT_BLOCK    DW      0                  ; Current block number
12      000E  0080                    RECORD_SIZE      DW      80H                ; Logical record size
13      0010  00 00 00 00             FILE_SIZE        DD      0                  ; Size of the file, in bytes
14      0014  0000                    DATE             DW      0                  ; Date the file was created
15      0016     0A [                 RESERVED         DB      10 DUP(?)          ; Reserved, can't be overriden
16                   ??
17                       ]
18
19      0020  00                      SEQ_NUMBER       DB      0                  ; Record in block for sequential
20      0021  00 00 00 00             RANDOM_NUMBER    DD      0                  ; Record in file for random
21      0025                          FCB     ENDS
22
23      0000                          STACK   SEGMENT STACK
24      0000     40 [                         DW      64 DUP(?)
25                   ????
26                         ]
27
28      0080                          STACK   ENDS
29
30      0000                          CODE    SEGMENT
31                                            ASSUME  CS:CODE
32
33
34      0000                          RECORDS         PROC    FAR
35      0000  1E                              PUSH    DS                          ; Set return address
36      0001  B8 0000                         MOV     AX,0
37      0004  50                              PUSH    AX
38                                            ASSUME  DS:CODE                     ; DS really points to PSP
39      0005  BA 005C                         MOV     DX,05CH                     ; Location of FCB in PSP
40      0008  B4 0F                           MOV     AH,0FH                      ; Open the file
41      000A  CD 21                           INT     21H
42
43      000C  BB 005C                         MOV     BX,05CH                     ; Location of FCB
44      000F  8B 47 14                        MOV     AX,[BX].DATE
45      0012  25 FE00                         AND     AX,MASK YEAR                ; Isolate the year
46      0015  B1 09                           MOV     CL,YEAR                     ; Right justify the value
47      0017  D3 E8                           SHR     AX,CL
48      0019  8A F0                           MOV     DH,AL                       ; Store year in BH
49
50      001B  8B 47 14                        MOV     AX,[BX].DATE
51      001E  25 01E0                         AND     AX,MASK MONTH
52      0021  B1 05                           MOV     CL,MONTH
53      0023  D3 E8                           SHR     AX,CL
54      0025  8A D0                           MOV     DL,AL                       ; Store month in BL
55
56      0027  8B 47 14                        MOV     AX,[BX].DATE
57      002A  25 001F                         AND     AX,MASK DAY                 ; Store day in AL
58
59      002D  CB                              RET
60      002E                          RECORDS         ENDP
61      002E                          CODE    ENDS
62                                            END     RECORDS
```

Abbildung 6.16 Strukturen und Records

DATE_WORD noch die FCB-Struktur im Programm aufruft. Sie dienen lediglich dazu, die Offsets der Datenbereiche zu definieren.

Ein abschließendes Wort über den Gebrauch von Records und Strukturen. Wir benutzen diese Mechanismen, weil sie uns das Schreiben eines Programmes ohne spezifische Kenntnis der Datenstruktur ermöglichen. Mit einer STRUC-Definition können wir jedes Feld als Offset innerhalb der Struktur ansprechen. Der Programmierer muß den tatsächlichen Offset eines Feldes nicht kennen. Dasselbe gilt für einen Bit-Datensatz. Wenn das Programm die MASK- und Shiftzähleroperatoren benutzt, muß es die Bit-Positionen im Feld nur in der RECORD-Anweisung spezifizieren.

Der Wert des Programmierens mit diesen Werkzeugen zeigt sich, wenn Sie an einem großen Projekt arbeiten, das viele Programmierer oder viele Programme hat. Stets werden Sie die Datenstrukturen modifizieren, wenn die Programme weiterentwickelt werden. Wenn Sie diese Datenstrukturmodelle beim Schreiben Ihres Programmes benutzen, ist es einfach, die Modelle und die zugehörigen Programme zu ändern. Sie können die Datenstruktur modifizieren und dann alle Programme, die diese Struktur benutzen, wieder assemblieren. Die Programme selbst müssen nicht geändert werden. Und darüber hinaus können Sie die Datenstruktur als eine getrennte Datei erhalten und sie in jeder Assemblierung mit INCLUDE einsetzen, so daß es immer nur eine einzige Version der Datenstruktur geben wird. Der Gebrauch dieser Werkzeuge vereinfacht also den Entwicklungsprozeß eines großen Programms, wenn es entwicklungbedingte Veränderungen erfährt.

7 Der 8087 Arithmetikprozessor

Die Entwickler des 8088 haben diesen Mikroprozessor mit einer speziellen Eigenschaft versehen, die einzigartig für die Rechnerfamilie 8086/8088 ist. Das Design des Prozessors erlaubt es einem System nämlich, über einen Coprozessor zu verfügen. Ein Coprozessor ist dabei ein Gerät, das die Eigenschaften des eigentlichen Prozessors in gewisser Hinsicht erweitert und ausdehnt. Der Arithmetikprozessor 8087 ist dabei als Coprozessor für den 8088 gedacht und verfügt über zusätzliche numerische Befehle und Gleitpunktregister. Diese zusätzlichen arithmetischen Fähigkeiten werden nun eine Erweiterung des 8088 und erhöhen die Verarbeitungsgeschwindigkeit erheblich, wenn wir Gleitpunktoperationen durchführen oder mit Zahlen von sehr hoher Genauigkeit arbeiten.

Der IBM PC ist bereits für den Einsatz eines Coprozessors vorgesehen. Dazu befindet sich auf der Systemplatine neben dem 8088 ein leerer 40-poliger Stecksockel. Der Sockel ist dabei so angeschlossen, daß jeder von der Architektur her passende Coprozessor für den 8088 verwendet werden kann. Der 8087 paßt dabei ganz genau, sowohl von der Architektur als auch rein physikalisch, in den Sockel. Um also den 8088 mit den gewünschten zusätzlichen numerischen Fähigkeiten zu versehen, brauchen wir nur ganz einfach den 8087 in den leeren Steckplatz einzustecken. Außerdem müssen wir auf der Systemplatine noch einen Schalter setzen, um dem Prozessor die Existenz des Coprozessors mitzuteilen. Nötig ist dieser Schalter in Wirklichkeit nur, um sicherzustellen, daß der 8088 die Interrupts für Fehlerbedingungen des 8087 erhält. Der tatsächliche Einsatz des 8087 wird nämlich nicht von irgendwelchen externen Schaltern beeinflußt.

Arbeitsweise des 8087

Der 8087 bearbeitet Gleitpunktbefehle durch Überprüfen der vom 8088 ausgeführten Anweisungen. Der Arithmetikprozessor sieht dabei dem 8088 bei der Befehlsausführung zu. Sieht der 8087 einen Befehl, den eigentlich er ausführen sollte, so greift er ein und beginnt nun seinerseits die Befehlsausführung. Der 8087 führt dabei den arithmetischen Befehl parallel zum 8088 aus. D.h., während der 8087 den arithmetischen Befehl ausführt, kann der 8088 bereits den nächsten Befehl ausführen. Auf diese Weise wird echte Parallelarbeit ermöglicht. Der 8088 kann dabei einen Befehl ausführen, während der 8087 einen ganz anderen Befehl bearbeitet. Dies kann besonders wertvoll werden, wenn die Abarbeitung eines Befehls auf dem 8087 längere Zeit in Anspruch nimmt, wie es beispielsweise für einige komplizierte Gleitpunktoperationen der Fall sein kann.

Da nun die beiden Prozessoren simultan arbeiten können, muß es einige Möglichkeiten der Synchronisation zwischen ihnen geben. Dazu wird der Befehl WAIT des 8088 verwendet. Der 8087 ist so mit dem 8088 zusammengeschaltet, daß der Arbeitszustand des Arithmetikprozessors (beispielsweise die Ausführung einer Gleitpunktoperation) die Testeingangsleitung des 8088 deaktiviert. Der WAIT-Befehl stoppt sodann den Befehlsablauf im 8088, bis die Testleitung aktiv wird und auf diese Weise mitteilt, daß der 8087 seinen Befehl abgearbeitet hat. Auf diese Weise kann der 8088 sicherstellen, daß der 8087 einen Befehl vollständig aus-

geführt hat, bevor er nun seinerseits wieder versucht, ihm einen weiteren Befehl durchzureichen. Außerdem wird auf diese Weise verhindert, daß der 8088 Zugriff auf irgendwelche vom 8087 gespeicherten Daten unternimmt, bevor dieser seinen Befehlszyklus abgearbeitet hat.

Prozessor und Coprozessor kommunizieren dabei nur über die externen Steuerleitungen des Prozessors, wie beispielsweise die Testsignalleitung. Der 8088 kann die internen Register des 8087 nicht lesen, ebenso auch umgekehrt. Alle Daten, die also zwischen diesen beiden Prozessoren ausgetauscht werden sollen, müssen im Speicher hinterlegt werden, auf den beide Prozessoren Zugriff haben. Da sich die Adressregister jedoch im 8088 befinden, wird es für den 8087 einigermaßen schwierig, den Speicher unter Verwendung der Adressierungsarten des 8088 effizient zu adressieren. Um es dem 8087 dennoch möglich zu machen, den Speicher mittels der Adressierungsmodi des 8088 anzusprechen, arbeiten die beiden Prozessoren bei der Ausführung einer Gleitpunktanweisung zusammen.

Der 8088 verfügt dazu über einen ESCAPE-Befehl (ESC), der bei ihm keinerlei Wirkung zeigt. In einem System, das auch einen 8087 enthält, entspricht der Befehl ESC dem Befehl NOP — mit der Ausnahme, daß er länger dauert. Allerdings führen die ESC-Befehle Adressinformationen mit sich. Genauer gesagt, sie führen ganz besonders ein mod-r/m Byte für die Adressberechnung als Teil des Befehlscodes mit sich. Obwohl nun der Befehl ESC wie ein NOP-Befehl wirkt, führt der 8088 die Adressberechnung durch. Der 8088 führt also mit der errechneten Adresse einen Lesebefehl im Speicher durch, obwohl er selbst mit diesen Daten nichts zu tun hat (wird dabei durch das mod-r/m Byte ein Register des 8088 anstelle eines Speicheroperanden angegeben, so erfolgt kein Speicherlesen).

In der Zwischenzeit hat der 8087 die vom 8088 ausgeführte Befehlsfolge mitbeobachtet. Führt nun der 8088 den Befehl ESC aus, so erkennt der 8087, daß es sich um einen Befehl für ihn handelt. Dazu wartet der 8087 nun noch den Dummy-Lesebefehl des 8088 ab. Liegt die so erzeugte Adresse auf dem Systembus, so übernimmt sie der 8087. Der 8087 weiß also auf diese Weise, wo sich die Daten im Speicher befinden, ohne selbst die Adresse errechnet zu haben. Der 8088 führt die Adressberechnung durch, der 8087 führt den Rest des Befehls aus. Im folgenden kann sich der 8087 dann einige Speicherzyklen stehlen, um Schreib- oder Leseoperationen mit den Daten auszuführen, die er benötigt, während der 8088 seinerseits den nächsten Befehl ausführt.

Der 8087 fügt dabei dem System einige zusätzliche arithmetische Fähigkeiten hinzu, ersetzt aber keinen der Befehle des 8088. Die Befehle ADD, SUB, MUL und DIV, wie wir sie in Kapitel 4 besprochen haben, werden weiterhin vom 8088 ausgeführt. Der arithmetische Coprozessor enthält nur zusätzliche und wesentlich mächtigere Befehle für den Umgang mit Zahlen. Vom Gesichtspunkt der Programmierung aus müssen wir ein System mit einem eingebauten 8087 als einen kompletten Prozessor betrachten, der nur über einen größeren Befehlssatz verfügt als der einfache 8088. Es bringt auch wenig, sich daran zu erinnern, welcher Prozessor nun welchen Befehl ausführt. Nur wenn nämlich der 8088 einen Befehl ausführt und direkt auf ein Ergebnis des 8087 wartet, also eine Synchronisierung mittels des

WAIT-Befehls notwendig ist, wird die Unterscheidung zwischen den einzelnen Prozessoren notwendig.

Es gibt allerdings ein Problem, wenn wir Programme für den 8087 schreiben und den IBM Makro-Assembler verwenden. Der Makro-Assembler enthält nämlich keinen Befehlscode für die Befehle des 8087. Um den 8087 also zu verwenden, müssen wir Befehle des 8087 bilden, und dazu den WAIT- und ESC-Befehl verwenden. Der beste Weg hierzu ist die Verwendung eines Satzes von Makros, die es uns erlauben, Befehle für den 8087 zu schreiben. In Kapitel 6 entwickelten wir bereits einige der Makros, die für den 8087 nötig sind. Sollten Sie vorhaben, den 8087 zu programmieren, so müssen Sie diese Beispiele auf den vollen Befehlssatz des 8087 erweitern.

Datentypen des 8087

Zur Unterstützung seiner erhöhten Leistung verfügt der 8087 über einen erweiterten Satz von Datentypen. Während der 8088 nur jeweils ein Byte oder ein Wort direkt als Operand verwenden kann, verfügt der 8087 über sieben verschiedene Typen von numerischen Daten. Sechs dieser neuen Datentypen sind dabei nur auf dem 8087 verfügbar. Abbildung 7.1 zeigt diese sieben verschiedenen Datentypen des 8087. Vier der verschiedenen Formate sind dabei für Integerzahlen gedacht, drei für Real- bzw. Gleitpunktzahlen. Eines der Integerformate wird anstelle für Binärarithmetik für das Rechnen mit erweiterten BCD-Zahlen verwendet.

In Abbildung 7.2 sehen wir, wie der 8087 die verschiedenen Zahlentypen im Speicher ablegt. Ebenso wie beim Datenformat des 8088 werden auch hier die niedrigwertigen Teile des jeweiligen Operanden an die niedrigste Speicheradresse gestellt. Das Vorzeichenbit erscheint immer im Byte mit der höchsten Speicheradresse. Die Bedeutung der einzelnen Felder werden wir dann später bei den verschiedenen Datentypen besprechen.

Der 8087 hat Zugriff auf drei Arten von Registern: Wörter, was 16 Bits bedeutet und dem Wortoperanden des 8088 entspricht; kurze Integerzahlen, ein 32-Bit Wert; und schließlich lange Integerzahlen, ein 64-Bit Wert. Alle Zahlen werden dabei im Zweierkomplement dargestellt.

In einem Programm legen wir eine Wortintegerzahl mit dem Operator DW fest. Die Zahl kann dann einen Wert zwischen −32768 und 32767 einnehmen. Diese Darstellung der Integerzahlen haben wir bereits im Befehlssatz des 8088 kennengelernt. Es ist im übrigen das einzige Datenformat, das 8088 und 8087 gemeinsam verwenden.

Eine kurze Integerzahl erfordert eine spezielle Angabe, um ein 32 Bit langes Datenfeld festzulegen. Dies erfolgt mit der Angabe DD (Define Doubleword), womit wir Integerzahlen im Bereich zwischen -2^{32} und $2^{32}-1$ festlegen können. Erinnern wir uns daran, daß wir mit der DD-Angabe auch das Adressenpaar SEGMENT:OFFSET bezeichnen können. Anhand des jeweils verwendeten Operanden stellt der Assembler nun fest, welche Form von Zahl er generieren soll. Ist der Operand dabei eine

Adresse, so wird das Adressdoppelwort SEGMENT:OFFSET erzeugt. Ist der Operand dagegen ein Zahlenwert, so wird die entsprechende Integerzahl erzeugt.

Zur Festlegung der 64 Bit langen Integerzahlen benötigen wir die Angabe DQ (Define Quadword). Diese Angabe teilt dem Assembler mit, daß er einen Datenbereich in der Länge von vier Worten (acht Bytes) erstellen soll. Mit dieser Art von Integerzahlen können wir einen Wert von -2^{64} bis $2^{64}-1$ darstellen. Mit der entsprechenden Assembleranweisung, genauso wie DB, DW und DD, können wir einen konstanten Datenwert ebenso erstellen wie mittels des Fragezeichens einen undefinierten Datenbereich oder über die Angabe DUP ein Mehrfaches eines acht Byte langen Datenfeldes.

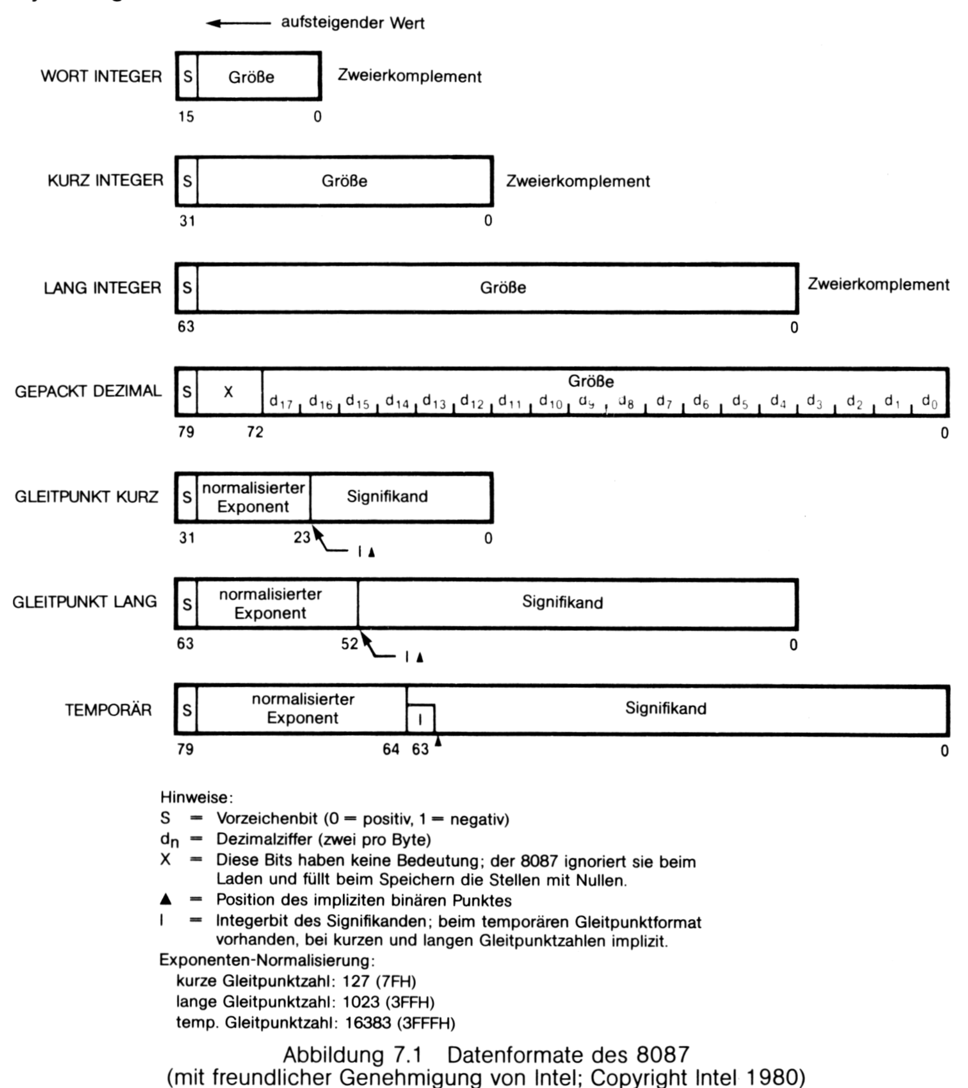


Abbildung 7.1 Datenformate des 8087
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

Den vierten Typ zur Darstellung von Integerzahlen verwenden wir für die Repräsentation von gepackten Dezimalzahlen. Dabei wird eine Integerzahl in Form eines gepackten BCD-Wertes dargestellt. Dazu benötigen wir zehn Bytes. Ein Byte ist dabei für das Vorzeichen reserviert, und die restlichen neun Bytes enthalten insgesamt 18 Dezimalstellen. Diese gepackte Datendarstellung entspricht der des 8088 mit der Ausnahme, daß wir hier mit einem Befehl 18 Stellen gleichzeitig verarbeiten können. Dezimalbefehle des 8088 erlauben uns dagegen nur die Verarbeitung von jeweils zwei Dezimalziffern. Die gepackten Dezimalwerte des 8088 verlangen vom Programmierer außerdem, die Art der Behandlung des Vorzeichens explizit festzulegen, sobald negative Zahlen verwendet werden. Die gepackten BCD-Zahlen bei der

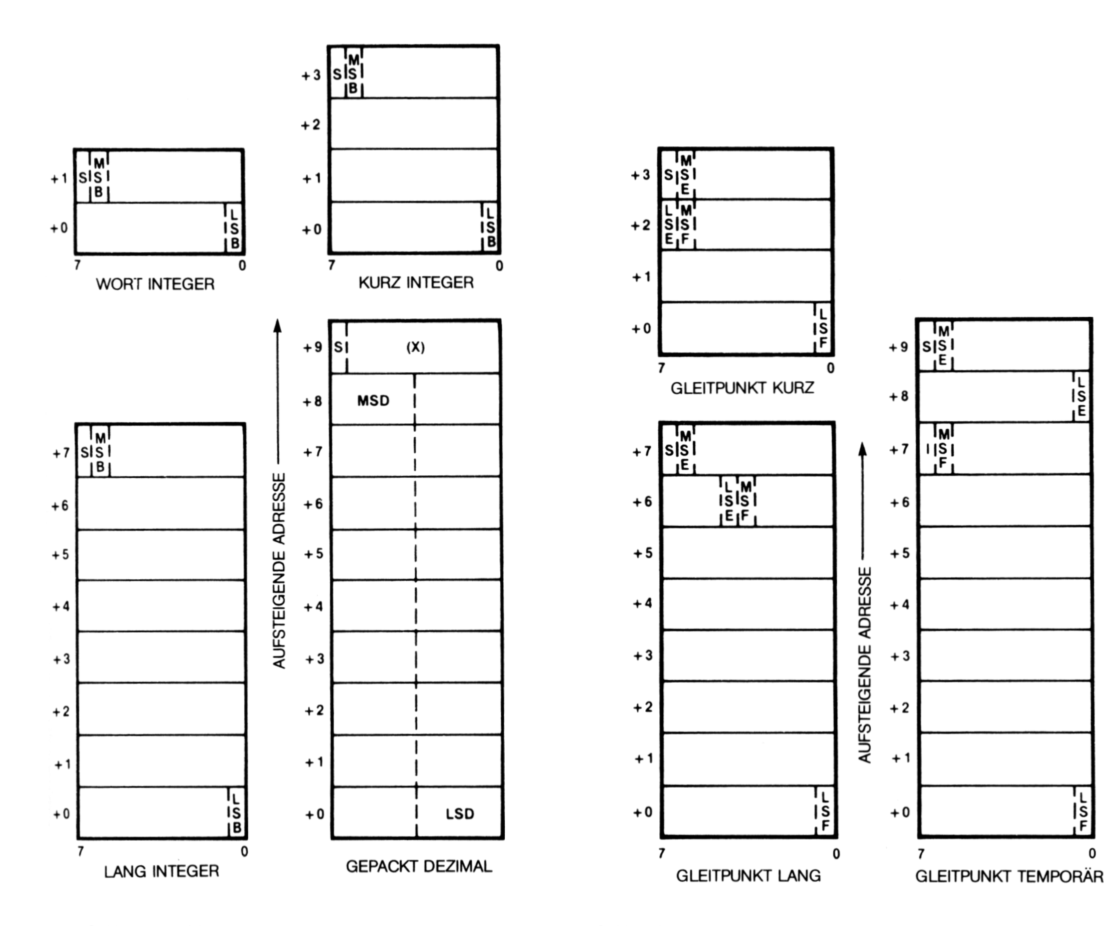
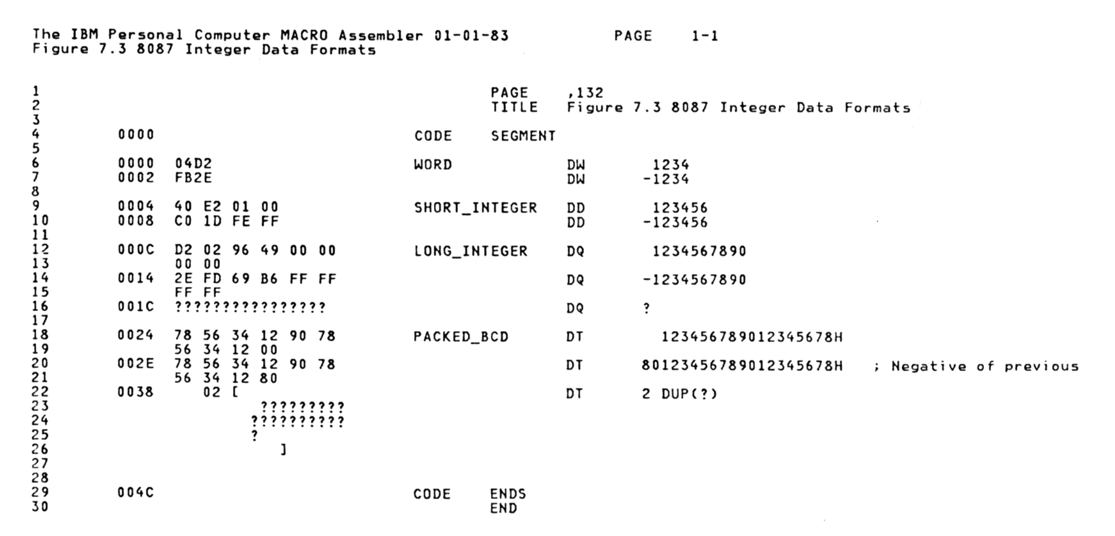


Abbildung 7.2 Speicherbelegung durch den 8087
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

Verarbeitung mit dem 8087 verfügen dagegen immer über ein Vorzeichenbit im höchstwertigen Byte. Der gepackte BCD-Wert wird dann vorzeichengerecht in dem 10-Byte Feld abgespeichert, wobei das höchstwertige Bit das Vorzeichen darstellt (0 entspricht positiv, 1 entspricht negativ).

Zur Definition eines solchen 10-Byte Datenbereichs zur Darstellung von gepackten Dezimalzahlen benützen wir die Assembleranweisung DT (Define Tenbyte). Mit dieser Anweisung reservieren wir einen 10-Byte Datenbereich. Um mit dem Assembler einen gepackten BCD-Wert in diesem Bereich festzulegen, müssen wir hexadezimale Notation verwenden. Würden wir nämlich eine normale Zahl verwenden, so würde sie der Assembler im Zweierkomplement darstellen und nicht als gepackte BCD-Zahl. Glücklicherweise ist es sehr einfach, eine Dezimalzahl in die benötigte hexadezimale Darstellung für gepackte BCD-Zahlen umzuwandeln. Dazu schreiben wir die Zahl nur ganz normal dezimal und fügen ihr am Ende ein H an, um die hexadezimale Darstellung zu kennzeichnen. Etwas schwieriger ist die Verwendung negativer Werte. Versehen wir nämlich die Dezimalzahl mit einem Minuszeichen, so setzt sie der Assembler in einen Wert im Zweierkomplementsystem um, selbst wenn wir ein H angeben. Um einen negativen gepackten BCD-Wert anzugeben, müssen wir deshalb die Dezimalstellen zählen und die Zahl auf insgesamt 20 Stellen erweitern. Die ersten beiden Stellen müssen dabei den Wert 80 enthalten, um damit das negative Vorzeichen darzustellen. Um im Assembler die gepackte BCD-Zahl −1234 darzustellen, müßten wir folgende Anweisung angeben:

```
DT      80000000000000001234H
```



```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 7.3 8087 Integer Data Formats

 1                                              PAGE    ,132
 2                                              TITLE   Figure 7.3 8087 Integer Data Formats
 3
 4        0000                          CODE    SEGMENT
 5
 6        0000  04D2                    WORD            DW       1234
 7        0002  FB2E                                    DW      -1234
 8
 9        0004  40 E2 01 00             SHORT_INTEGER   DD       123456
10        0008  C0 1D FE FF                             DD      -123456
11
12        000C  D2 02 96 49 00 00       LONG_INTEGER    DQ       1234567890
13              00 00
14        0014  2E FD 69 B6 FF FF                       DQ      -1234567890
15              FF FF
16        001C  ????????????????                        DQ      ?
17
18        0024  78 56 34 12 90 78       PACKED_BCD      DT        12345678901234567 8H
19              56 34 12 00
20        002E  78 56 34 12 90 78                       DT      80123456789012345678H   ; Negative of previous
21              56 34 12 80
22        0038     02 [                                 DT      2 DUP(?)
23                         ?????????
24                        ??????????
25                        ?
26                           ]
27
28
29        004C                          CODE    ENDS
30                                              END
```

Abbildung 7.3 8087 Integer-Datenformate

In Abbildung 7.3 sehen wir eine Assemblerliste, die die vom Assembler generierten Werte für die vier verschiedenen Integertypen darstellt.

Darstellung von Gleitpunktzahlen

Es gibt drei Formate für die Darstellung von realen Zahlen im 8087. Zwei der Darstellungsarten für Gleitpunktzahlen entsprechen dabei dem vorgeschlagenen Standard der IEEE für solche Zahlen. Das kurze Format verwendet 32 Bits, das lange Format 64. Das dritte Format spezifiziert ein 80 Bit langes Feld als Zahl. Dieses Format entspricht also nicht dem IEEE-Standard. Der 8087 verwendet dieses Zwischenformat zur internen Speicherung von Zwischenergebnissen, um eine möglichst hohe Rechengenauigkeit zu ermöglichen.

Der Rest dieses Abschnitts ist für die gedacht, die keine Erfahrung mit der Darstellung von Gleitpunktzahlen auf einem Rechner haben. Sie können diesen Abschnitt überspringen, wenn Sie bereits eine Vorstellung von der allgemeinen Darstellungsweise von Gleitpunktzahlen sowohl auf Papier als auch innerhalb eines Rechners haben. Der nächste Abschnitt behandelt die spezielle Art der Darstellung von Gleitpunktzahlen im 8087.

Integerdarstellung ist die beste Art der Zahlendarstellung für viele Arten von Zahlen. Die Darstellung von Integerzahlen ist einfach, sowohl im Verständnis als auch in der Anwendung, und paßt außerdem sehr gut zur binären Datendarstellung. Allerdings ist die Integerdarstellung nicht sehr geeignet für die Darstellung von großen Werten. Eine sehr große Integerzahl endet nämlich normalerweises mit einer langen Reihe von Nullen. Ein Beispiel für eine solche Zahl: Die Sonne ist ungefähr 93.000.000 Meilen von der Erde entfernt. Außerdem können wir mit der Integerdarstellung keine Zahlen abbilden, die nicht vollständig aus ganzen Zahlen bestehen. Ein Computer kann beispielsweise die Zahl $\frac{1}{2}$ nicht in der Integerdarstellung abbilden. Ebenso können wir Brüche, die kleiner als 1 sind, so wie $\frac{1}{3}$ oder $\frac{1}{5}$, in der Integerdarstellung nicht abbilden.

Wissenschaftler und Mathematiker haben bereits seit langer Zeit einen Weg entwickelt, der auf bestechende Weise das Problem der Darstellung von solchen Zahlen löst. Der erste Schritt ist dabei die Einführung eines Dezimalpunkts. Dieses Zeichen stellt die Trennung zwischen dem ganzzahligen Teil und dem Bruchteil einer Zahl dar. Bei einer Integerzahl, also einer ganzen Zahl, befindet sich der Dezimalpunkt immer an der äußersten rechten Seite der Zahl. Wird der Dezimalpunkt dagegen verwendet, so stellen die Zahlen rechts von diesem Dezimalpunkt immer einen Wert kleiner als 1 dar.

Bei Integerzahlen entspricht außerdem jede Ziffernposition einer Potenz von 10. So ist die Zahl 1234 also

$$1234 = 1000 + 200 + 30 + 4 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

Der Dezimalpunkt steht also nun genau an der Grenze zwischen der Position 10^0 und dem nachfolgenden gebrochenen Teil der Zahl. Auch in diesem gebrochenen Teil der Zahl, also hinter dem Dezimalpunkt, entsprechen die einzelnen Ziffernpositionen wiederum Potenzen von 10, aber in diesem Fall den negativen Potenzen von 10. Die Zahl 1.234 ist also

$$1.234 = 1 + 0.2 + 0.03 + 0.004 = 1 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3}$$

Der Dezimalpunkt erlaubt es uns also, Brüche darzustellen. Die Zahl $1\frac{1}{2}$ entspricht also 1.5, während ein Viertel gleich 0.25 und ein Fünftel gleich 0.2 ist.

Da jede Zahl im dezimalen Zahlensystem sich von ihrer Nachbarzahl um einen Faktor von 10 unterscheidet, entspricht auch eine Multiplikation von 10 einem Verschieben des Dezimalpunkts um eine Position nach rechts. Entsprechend wäre eine Division durch 10 ein Verschieben des Dezimalpunkts um eine Stelle nach links. Wir können dies Kenntnis verwenden, um den Dezimalpunkt an die geeignete Position zu schieben. Wir verschieben den Dezimalpunkt und verändern dabei gleichzeitig die Zahl jeweils um den Faktor 10. Dies wird gemeinhin als Gleitpunktdarstellung bezeichnet, da der Dezimalpunkt innerhalb der Zahl gleiten kann. Die Position des Dezimalpunkts als Trennung zwischen ganzen Zahlen und gebrochenen Zahlen liegt nicht länger absolut fest. Wir können also die Position dieses Dezimalpunkts entsprechend unseren Wünschen wählen und dann jeweils die Zahl mit der korrekten Potenz von 10 multiplizieren, um wieder den originalen Wert zu erhalten.

Beispielsweise ist die Sonne 93.000.000 Meilen von der Erde entfernt. Diese Zahl ist vor allem wegen ihrer großen Anzahl von Nullen schlecht zu handhaben. Dagegen können wir die Zahl in der Gleitpunktdarstellung ganz einfach als 9.3×10^7 schreiben. Das heißt, 93.000.000 entspricht vollkommen 9.3 mal 10 in der siebten Potenz, und in der Tat ist

$$\mathbf{93.000.000 = 9.3 \times 10^7 = 93 \times 10^6 = 930 \times 10^5}$$

usw. Wir können also den Dezimalpunkt an jede Position innerhalb der Zahl verschieben, indem wir die jeweilige Potenz von 10 verändern.

Eine dezimale Gleitpunktzahl hat also zwei Komponenten. Die eine ist der signifikante Teil der Zahl, und wird als Mantisse oder Signifikand bezeichnet. In unserem Beispiel war der Wert der Mantisse 9.3. In der Praxis nimmt die Mantisse einen Wert zwischen $1 \leqq \text{Mantisse} < 10$ ein. So könnte ein Wert für die Mantisse beispielsweise 1.3, oder 7.6 oder 9.97 sein. Die andere Komponente einer Gleitpunktzahl ist der Exponent. Dies ist die Potenz, in die 10 erhoben werden muß, bevor wir damit die Mantisse multiplizieren. So hat die Zahl 9.3×10^7 eine Mantisse von 9.3 und einen Exponenten von 7. Liegt dabei die Basis des Zahlensystem einmal fest, in unserem Fall 10, sind Mantisse und Exponent die einzigen beiden Werte, die bekannt sein müssen, um die Originalzahl wieder herzustellen.

Diese Gleitpunktdarstellung erlaubt es uns, sehr große Zahlen (beispielsweise 1.234×10^{85}), oder sehr kleine Zahlen (beispielsweise 1.234×10^{-85}) in einer kompakten Weise darzustellen. Das Schreiben dieser Zahlen ohne die Darstellung mit dem Exponenten von 10 würde im allgemeinen eine sehr lange Reihe von Nullen erfordern.

Die binäre Gleitpunktdarstellung ist der dezimalen Gleitpunktdarstellung sehr ähnlich. Die Basis des Zahlensystems ist dann ganz einfach 2 anstelle von 10. Die Mantisse ist entsprechend ein Wert von $1 \leqq \text{Mantisse} < 2$, und der Exponent ist eine Potenz von 2. So bedeutet, in Binärdarstellung, 1.101×10^{100}, daß die Mantisse 1.101B mit dem Wert 2^4 multipliziert wird, also mit 16. Der Wert der Mantisse wird dabei durch die gleiche Positionsbeziehung festgelegt wie bei Dezimalzahlen, mit

der Ausnahme, daß die Basis des Zahlensystems nun 2 ist. Zur Rechten des binären (Dezimal-)Punkts werden dann eben negative Potenzen von 2 dargestellt. Die Tabelle in Abbildung 7.4 zeigt die Werte der ersten fünf Positionen.

Wir sind nun in der Lage, den dezimalen Wert unserer Beispielszahl zu berechnen.

$$1.101B = 1 + \tfrac{1}{2} + \tfrac{1}{8} = 1\tfrac{5}{8} = 1.625$$

$$10^{100B} = 2^4 = 16$$

$$1.101 \times 10^{100B} = 1\tfrac{5}{8} \times 16 = 26$$

Alternativ hierzu könnten wir den Zahlenwert genau so berechnen, wie wir es für Dezimalzahlen tun. Der Exponent sagt uns dabei, um wieviele Positionen wir den Binärpunkt verschieben müssen. In unserem Falle, da der Exponent 4 ist, werden wir den Binärpunkt um 4 Stellen nach rechts verschieben.

$$1.101 \times 10^{100B} = 11010B = 26$$

Beide Methoden sind zulässig und erzeugen auch die gleichen Ergebnisse. Im ersten werteten wir zuerst den Bruch aus und multiplizierten ihn dann mit dem Exponenten. Im zweiten multiplizierten wir zuerst den Exponenten, bevor wir die Mantisse auswerteten.

binärer Wert	dezimaler Wert
0.1	1/2
0.01	1/4
0.001	1/8
0.0001	1/16
0.00001	1/32

Abbildung 7.4 Negative Potenzen von 2

Wie werden nun diese beiden Zahlen in einem Rechner dargestellt? Der Speicherplatz, der für die Aufnahme von Gleitpunktzahlen vorgesehen ist, wird in zwei Teile geteilt. Ein Teil enthält dabei das Vorzeichen und den Wert der Mantisse. Der andere Teil enthält das Vorzeichen und den Wert des Exponenten.

Die Größe des Mantissenfeldes bestimmt dabei die Präzision oder Genauigkeit der Zahl. Je mehr Bits der Rechner für das Mantissenfeld zuweist, desto höher wird die Genauigkeit. Beispielsweise ist der Dezimalbruch 1.234 wesentlich genauer als der Bruch 1.2. Die beiden zusätzlichen Stellen nach dem Dezimalpunkt erlauben eine genauere Bestimmung des eigentlichen Wertes.

Um nun eine möglichst große Genauigkeit zu erreichen, speichern beinahe alle Rechner die Mantisse als normalisierte Zahl. Dies bedeutet, daß die Mantisse eine Zahl zwischen eins und zwei ($1 \leqq \text{Mantisse} < 2$) ist. Zwei Überlegungen machen dies notwendig. Erstens, keine Art von führenden Nullen erhöht die Genauigkeit einer Zahl. (Dies trifft jedoch nicht für die Nullen am Schluß einer Zahl zu. Wir gehen nämlich davon aus, daß die Zahl 1.000 wesentlich genauer ist als die Zahl 1.0). Würden wir führende Nullen in der Mantisse einer Gleitpunktzahl zulassen, würde dies die Genauigkeit der Zahl verringern. Zweitens verlangt das Speichern einer Gleitpunktmantisse, daß sich der Binärpunkt an einer festen Position befindet. Und

in der Tat, der Binärpunkt der Mantisse befindet sich immer nach der ersten Binärzahl. Das Normalisieren der Mantisse erzwingt also eine 1 in der ersten Bitposition, wodurch sich der Wert der Mantisse zwischen eins und zwei bewegen kann. Der Computer richtet nun den Exponententeil der Zahl auf den entsprechenden Wert aus, um ihn mit der bereits normalisierten Mantisse abzustimmen.

Es gibt allerdings bei der Normalisierung der Mantisse Ausnahmen. Die augenfälligste Ausnahme ist dabei, wenn der Wert 0 auftritt. In diesem Fall ist die gesamte Mantisse 0. Die zweite Ausnahme ist nicht so augenscheinlich und tritt auf, wenn sich die dargestellte Zahl der unteren Grenze der Darstellbarkeit nähert. Sehen wir uns dies einmal an.

So wie die Größe der Mantisse einer Zahl ihre Präzision festlegt, so bestimmt die Größe des Exponentenfeldes die Größe der darstellbaren Zahl. Das Exponentenfeld enthält dabei jeweils eine Potenz von 2. Je mehr Bits sich nun in diesem Feld befinden, umso größer (oder kleiner) kann die Zahl werden.

Sind z.B. für den Exponenten drei Bits vorgesehen (dargestellt im Zweierkomplement), so ist die größte darstellbare Zahl 1.111×10^{011B}. Die Mantisse wäre in diesem Fall geringfügig kleiner als 2, während der Exponent 2^3, also 8, ist. So ist die maximal darstellbare Zahl geringfügig kleiner als 16. Die kleinste darstellbare positive Zahl wäre dann 1.000×10^{100B} oder 1×2^{-4}, also $\frac{1}{16}$. Da bei der Darstellung von Gleitpunktzahlen das Vorzeichen in der Mantisse nicht berücksichtigt wird, gelten für die Darstellbarkeit einer negativen Zahl dieselben Grenzen. Mit einem 3-Bit Exponenten laufen die positiven Zahlen von $\frac{1}{16}$ bis 16 und die negativen Zahlen von -16 bis $-\frac{1}{16}$.

Verwenden wir nun vier Bits für das Exponentenfeld, so ist die größte darstellbare Zahl $1.111 \times 10^{0111B} < 2 \times 2^7 = 256$. Die kleinste darstellbare positive Zahl wäre dann $1.000 \times 10^{1000B} = 1 \times 2^{-8} = \frac{1}{256}$. Die vier Exponentenbits würden uns also einen Zahlenbereich von $\frac{1}{256}$ bis 256 ermöglichen. Wir sehen, daß mit steigender Anzahl der Exponentenbits auch die Größe der darstellbaren Zahl zunimmt.

Es ist jedoch wichtig, festzuhalten, daß durch eine Steigerung der Anzahl der Bits des Exponenten zwar die Größe der darstellbaren Zahl ansteigt, die Genauigkeit jedoch nicht. In unserem obigen Beispiel haben wir angenommen, daß die Mantisse jeweils vier Bits lang ist. Wenn wir nun eine beliebige Zahl nehmen, die innerhalb des Bereichs der beiden Beispiele bleibt, so wird sie immer die gleiche Genauigkeit aufweisen ohne Rücksicht auf den jeweiligen Exponenten. So ist beispielsweise $1.010 \times 10^{001B} = 2\frac{1}{2}$. Eine Erweiterung der Anzahl der Exponentenbits führt nicht zu einer Erhöhung der Genauigkeit dieser Zahl. Für jede beliebige Zahl innerhalb der erlaubten Grenzen dieses Systems bestimmt allein die Mantisse die Genauigkeit.

Exponentenwerte werden nicht im Zweierkomplement gespeichert. Aus Gründen der Vereinfachung für die weitere Verarbeitung durch den Computer wird der Exponent als normalisierte Zahl gespeichert. Dies bedeutet, daß vor dem Speichern des aktuellen Exponenten auf diesen ein Ausgleichswert addiert wird. Dieser Ausgleichswert ist dabei so gewählt, daß es möglich ist, Exponenten mittels einfacher vorzeichenloser arithmetischer Befehle zu vergleichen. Dies ist besonders beim Vergleich zweier Gleitpunktzahlen von Vorteil. Exponent und Mantisse sind nämlich

als eine Dateneinheit gespeichert, wobei der Exponent der Mantisse vorausgeht. Ist der Exponent nun normalisiert, so kann ein Programm die beiden Zahlen bitweise, vom höchstwertigen zum niederwertigsten Bit, vergleichen. Bereits das erste Auftreten einer Ungleichheit legt die Reihenfolge der beiden Zahlen fest. Dann müssen keine weiteren Zahlenteile mehr überprüft werden. Der verwendete Normalisierungswert ist dabei durch die Größe des verwendeten Exponentenfelds bestimmt. Nehmen wir ein Beispiel aus den Formaten des 8087.

Eine kurze Gleitpunktzahl im 8087 ist 32 Bits lang, von denen 8 Bits für den Exponenten verwendet werden. Der Exponent kann also einen Wert von −128 bis +127 einnehmen. Außerdem ist beim 8087 der Wert −128 für eine spezielle unendliche Zahl reserviert, so daß sich der tatsächliche Exponent im Bereich von −127 bis 127 bewegt. Dieser Exponent wird nicht im Zweierkomplement dargestellt. Der 8087 addiert den Normalisierungswert 127 auf den Exponenten, bevor er ihn im Gleitpunktformat speichert. Die Tabelle im Abbildung 7.5 zeigt einige der möglichen Exponentenwerte, und die sich daraus ergebenden gespeicherten, tatsächlichen Werte.

Wert des Exponenten	normalisierter Wert
−127	000H
−1	07EH
0	07FH
1	080H
127	0FEH

Abbildung 7.5 Normalisierte Exponenten

Wie wir aus der Tabelle in Abbildung 7.5 ersehen, richtet der Normalisierungswert das Exponentenfeld so aus, daß der kleinste Exponent auch der kleinsten Zahl entspricht. Entsprechend gehört zum größten Exponenten auch die größte Zahl. Unterscheiden sich also zwei Gleitpunktzahlen in ihrem Exponentwert, so genügt ein einfacher Vergleich der beiden Exponenten, um eine Reihenfolge der Zahlen herzustellen. Da dieser Vergleich auch über das Mantissenfeld weiterläuft, kann ein Programm sehr einfach die Reihenfolge von zwei Gleitpunktzahlen feststellen. Und diese einfache Vorgehensweise wird erst durch die Normalisierung des Exponenten ermöglicht.

Formate für reale Zahlen im 8087

Es gibt drei Gleitpunktdatenformate, die vom 8087 unterstützt werden. Diese Formate sehen wir im Abbildung 7.1 und 7.2. Abbildung 7.1 zeigt dabei den logischen Aufbau der Zahlen, während Abbildung 7.2 die Position der einzelnen Komponenten bei einem im Speicher abgelegten Wert zeigt.

Die Formate für lange und kurze Realzahlen entsprechen dabei dem IEEE-Standard für Gleitpunktzahlen. Eine kurze Realzahl benötigt dabei 32 Bits, oder 4 Bytes. Dieses Format wird oft auch als einfach genaue Gleitpunktzahl bezeichnet. Die Mantisse enthält 23 Bits, was einer Genauigkeit von etwa 6 bis 7 Dezimalziffern entspricht. D.h., eine siebenstellige Dezimalzahl ist ungefähr so genau wie eine 23 Bit

lange Binärzahl. Der 8 Bit lange Exponent hat einen Normalisierungswert von 127 oder 07FH. Der Exponent weist dadurch einen möglichen Darstellungsbereich von 2^{-127} bis 2^{127} auf, was einem Zahlenbereich von 10^{-38} bis 10^{38} entsprechen würde. Das verbleibende Bit bestimmt das Vorzeichen der gesamten Zahl. Halten wir fest, daß es innerhalb einer Gleitpunktzahl zwei Vorzeichenbits gibt. Eines davon ist das Vorzeichen für den Exponenten, und im Exponentenfeld auch enthalten (und wird zusätzlich durch den Normalisierungswert verändert). Das zweite Vorzeichenbit gibt an, ob die Zahl selbst insgesamt positiv oder negativ ist.

Das lange Format für Realzahlen benötigt 64 Bits oder 8 Bytes. Diese doppelt genaue Gleitpunktzahl hat eine Mantisse von 52 Bits, was etwa 15 bis 16 Dezimalstellen entspricht. Der 11 Bit lange Exponent hat einen Darstellungsbereich von 2^{-1023} bis 2^{1023} und einen Normalisierungswert von 1023 oder 03FFH. In Dezimalzahlen ausgedrückt, liegt der darstellbare Wert zwischen 10^{-307} und 10^{307}.

Kurze oder lange reale Zahlen werden vom 8087 immer als normalisierte Zahlen gespeichert. Noch einmal sei gesagt, daß dies bedeutet, daß das erste Bit der Mantisse eine 1 enthält. Da dieses Bit immer 1 ist, gibt es auch keinen Grund mehr, es überhaupt abzuspeichern. Es wird einfach vorausgesetzt, daß es immer da ist. In Abbildung 7.2 können wir dies sehen. Unterhalb des Datenbereichs für lange und kurze Gleitpunktzahlen geht eine 1 dem binären Punkt voraus. Dieses Bit befindet sich zwar nicht im Speicher, wir wissen jedoch, daß es sich hier befindet.

Wir sollten allerdings festhalten, daß die sechs Datenformate des 8087, die wir bisher besprochen haben — vier Integer- und zwei Gleitpunktformate — nur außerhalb des 8087 existierten. D.h., der 8087 verwandelt die Daten nur dann in dieses Format, wenn er sie im Speicher ablegen muß. Der 8087 kann die Daten allerdings auch in diesem Format lesen. Alle Daten innerhalb des 8087 befinden sich dagegen im siebten Format — dem sogenannten temporären Gleitpunktformat. D.h., daß Programme, die wir schreiben, um den 8087 zu benützen, sich den Vorteil des erweiterten Zahlenbereichs und der größeren Genauigkeit dieser temporären Gleitpunktzahlen zunutze machen können, solange sie nur innerhalb des 8087 verarbeitet werden. Die externen Datenformate sind für uns als Arbeitserleichterung gedacht, wenn wir Daten speichern oder lesen wollen.

Die temporäre Gleitpunktdarstellung ist dabei eine Darstellung mit größtmöglicher Genauigkeit und mit dem größten Zahlenbereich. Diese Darstellungsart benötigt 80 Bits oder 10 Bytes. Die Mantisse ist dabei 64 Bits lang. Dies entspricht einer Genauigkeit von ungefähr 19 Dezimalstellen. Die Mantisse ist im allgemeinen normalisiert, kann jedoch unter bestimmten Bedingungen auch denormalisiert werden. Aus diesem Grund wird beim temporären Gleitpunktformat auch nicht davon ausgegangen, daß das höchstwertige Bit der Mantisse immer eine 1 enthält. In Abbildung 7.2 sehen wir die normalisierte führende 1 sehr genau als Teil der Mantisse, und nicht als angenommenen Wert. Das 16 Bit lange Exponentenfeld wird mit einem Wert von 16383 oder 03FFFH normalisert. Der Exponent kann dabei einen Wert von $2^{-16,383}$ bis $2^{16,383}$, also 10^{-4932} bis 10^{4932} einnehmen.

Da der 8087 zusätzlich die Mantisse einer temporären Gleitpunktzahl denormalisieren kann, liegt die untere Grenze einer solchen Zahl noch niedriger. Wird die Zahl

nämlich denormalisiert, so erscheinen führende Nullen in dem gebrochenen Teil der Zahl. Dies erlaubt die Darstellung einer Zahl, die noch kleiner ist, als es der Bereich des Exponenten erlauben würde. Sehen wir uns zu diesem Beispiel die einfache Form an, die wir früher verwendeten, also einen 3-Bit Exponenten und eine 4-Bit Mantisse. Die kleinste damit darstellbare Zahl wäre

$$1.000 \times 10^{000B} = 1 \times 2^{-3} = \frac{1}{8}$$

Dieses Beispiel setzt voraus, daß der Exponent mit einem Wert von 3 normalisiert wird. Denormalisieren wir nun die Mantisse, so können wir sogar eine noch kleinere Zahl darstellen, wie beispielsweise

$$0.100 \times 10^{000B} = \frac{1}{2} \times 2^{-3} = \frac{1}{16}$$

Die kleinste positive Zahl, die wir auf diese Weise mit einem denormalisierten Wert darstellen könnten, wäre

$$0.001 \times 10^{000B} = \frac{1}{8} \times 2^{-3} = \frac{1}{64}.$$

Durch Denormalisieren der Mantisse erweitern wir also die untere Grenze des darstellbaren Zahlenbereichs.

Dieser zusätzliche Bereich ist natürlich nicht umsonst. Als wir führende Nullen einführten, haben wir damit automatisch die Genauigkeit der Mantisse verringert. Das temporäre Gleitpunktformat des 8087 erlaubt es uns, auf Kosten der Präzision den darstellbaren Zahlenbereich zu vergrößern, falls dies nötig sein sollte. Diese Möglichkeit kommt hauptsächlich in Zwischenschritten bei langen Berechnungen zum Tragen. Es kann nämlich vorkommen, daß vor Ausführen einer anderen Operation zwei beinahe gleichgroße Zahlen voneinander subtrahiert werden. Das Ergebnis dieser Subtraktion wäre eine Zahl, die wesentlich kleiner ist als jede der beiden Originalzahlen. Das Ergebnis könnte jedoch sehr bedeutsam sein. Vergrößern wir in diesem Fall den darstellbaren Zahlenbereich, kann der Computer mit seiner Berechnung fortfahren. Die einzige andere Möglichkeit wäre, das Ergebnis der Subtraktion auf Null zu setzen. Und in diesem Falle wäre dann die ganze Bedeutung der Berechnung verloren.

Definition von Gleitpunktzahlen

Der Assembler reserviert Datenbereiche für Gleitpunktzahlen mit einer Länge von 4, 8 oder 10 Bytes. Für kurze Gleitpunktzahlen verwenden wir dabei die Angabe DD. Für lange Gleitpunktzahlen verwenden wir die Angabe DQ, und für temporäre Gleitpunktzahlen die Angabe DT.

Unglücklicherweise unterstützt der IBM Makro-Assembler die Gleitpunktdarstellung für den 8087 nicht. Geben wir also als Operand in einer DD oder DQ-Anweisung eine Gleitpunktzahl an, so wird das Ergebnis eine unentzifferbare Folge von Nullen und Einsen sein. Das so erzeugte Gleitpunktformat entspricht nämlich dem Datenformat, das der Basic-Interpreter intern verwendet und entspricht in keiner Weise dem IEEE-Standardformat, das der 8087 verwendet. So müssen wir also in einem Programm für den 8087 eine andere Methode verwenden, um Gleitpunktzahlen anzugeben.Es gibt zwei Methoden, Gleitpunktzahlen zu erzeugen. Der erste

Weg ist dabei der menschliche Assembler. Das bedeutet, daß wir mit Bleistift und Papier die Gleitpunktzahl ausrechnen und das Ergebnis an die passende Stelle legen. Dieses Vorgehen bedeutet unter anderem, daß wir das Format sehr genau überprüfen, die im allgemeinen dezimal vorliegende Zahl in das korrekte Binärformat überführen, Exponent und Mantisse trennen, den Exponenten normalisieren und schließlich das Ergebnis in binäre oder hexadezimale Konstanten verwandeln müssen. Dies ist ein gangbarer Weg, doch ein Weg, der einen großen Aufwand an Geschick und auch einen ganz schönen Zeitaufwand bedeutet.

Der zweite Weg, Gleitpunktzahlen zu erzeugen, ist die indirekte Methode, die zwar bei der Ausführung einen größeren Zeitaufwand benötigt, aber dafür umso weniger Zeit beim Programmieren. Wir schreiben in diesem Fall die Gleitpunktzahl als Produkt oder Quotient von zwei oder mehreren Integerzahlen. Der Assembler kann die Integerzahlen dabei direkt verarbeiten, wenn wir die Anweisungen DW, DD oder DQ verwenden. Nach der Initialisierung des 8087 kann unser Programm dann die gewünschten Gleitpunktzahlen aufbauen, indem wir die benötigten Multiplikationen oder Divisionen durchführen. Nehmen wir als Beispiel einmal an, daß unser Programm die Gleitpunktzahl 1.234×10^5 als Konstante im Speicher benötigt. Als Integerzahl können wir dies ganz einfach darstellen.

```
DD      123400
```

Halten wir dabei fest, daß die Zahl zu groß für eine Darstellung in einem DW-Feld ist, daß sie aber ohne Schwierigkeiten die Bedingungen für eine 32-Bit Integerzahl erfüllt.

Als weiteres Beispiel könnten wir annehmen, daß wir die Zahl 1.234×10^{-5}, also eine sehr kleine Zahl, in unserem Programm benötigen. In diesem Falle benötigen wir zwei Integerzahlen. Eine Integerzahl entspricht dabei der Mantisse, die andere dem Exponenten. In unserem Programm können wir dann die folgenden Definitionen verwenden, um die Gleitpunktzahl zu erzeugen.

```
DW      1234
DD      100000000
```

Wir dividieren den gebrochenen Teil der Zahl, also 1234, durch den Exponenten, 10^8, um die eigentliche Gleitpunktzahl zu erzeugen.

Ein Problem bei dieser Arbeitsweise stellen sehr kleine oder sehr große Zahlen dar. Diese Zahlen benötigen nämlich Exponenten, die wesentlich größer sind als alle Integerzahlen, die wir darstellen können. Für diesen Sonderfall werden wir später ein Beispiel geben, das das Gleitpunktformat für jede dritte Potenz von 10 darstellt. Die Besprechung dieses Programms schieben wir aber auf, bis wir die dazu nötigen Befehle besprochen haben. Mit diesen Gleitpunktdarstellungen für Potenzen von 10 wird es für uns ein leichtes sein, Werte wie z.B. 10^{36} oder 10^{-24} darzustellen. Unser Programm kann dann die benötigte Gleitpunktzahl in einen Integerteil und eine Potenz von 10 aufbrechen. Multiplikation oder Division erzeugen den korrekten Wert. Der 8087 bietet uns zusätzlich noch andere Möglichkeiten, solche Probleme zu lösen, doch die Methode der Benutzung von Integerzahlen und Potenzen von 10 ist die einfachste, speziell für Leute, die den 8087 zum ersten Mal benützen.

Programmieren mit dem 8087

Obwohl der 8087 ein eigener Prozessor ist, sollten wir ihn als eine Erweiterung des 8088 betrachtet, der diesem zusätzliche Fähigkeiten verleiht. Diese Fähigkeiten sind zusätzliche Datentypen, zusätzliche Befehle und zusätzliche Register. Abbildung 7.6 zeigt ein Beispiel für die Programmierung des 8087. Die Register des 8087 bilden zusammen mit denen des 8088 einen kompletten Registersatz für den Programmierer.

Registerstack

Der 8087 verfügt über vier besondere Register und einen acht Register langen Stack, um die numerischen Operanden aufzunehmen. Da wir mit dem 8087 hauptsächlich über den Registerstack arbeiten, werden wir ihn als erstes besprechen.

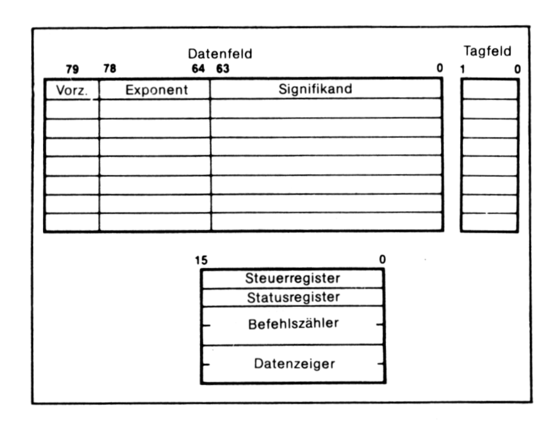


Abbildung 7.6 Register des 8087
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

Der Registerstack des numerischen Coprozessors verfügt über acht Positionen, von denen jede 80 Bits breit ist. Der 8087 speichert dabei alle zu verwendenden Daten im temporären Gleitpunktformat ohne Rücksicht auf ihre ursprüngliche Darstellung im Speicher. Dabei werden vom 8087 sowohl Integer- als auch Gleitpunktzahlen in das interne Format verwandelt. Wie wir bereits bemerkt haben, erlaubt uns diese Darstellung sehr große Präzision und außerdem einen sehr großen Zahlenbereich. Für Rechenoperationen mit Integerzahlen erhalten wir mit dem 8087 exakte Ergebnisse bis zu einer Größenordnung von 2^{64}. Die interne Gleitpunktdarstellung können wir allerdings außer acht lassen, wenn wir uns mit Integerarithmetik auf dem 8087 beschäftigen.

Die Register des 8087 arbeiten als Push-Down-Stack, ebenso wie der Stack des 8088. Im Gegensatz zu diesem sind allerdings nur eine beschränkte Anzahl von Plätzen vorhanden, nämlich genau acht. Der 8087 verfügt außerdem über ein zusätzliches Register innerhalb des Prozessors, das vom Programmierer nicht ganz

so einfach angesprochen werden kann. Dieses Register enthält ein Belegungswort für jede Position auf dem Stack. Auf diese Weise kann der 8087 einfach feststellen, welche Positionen des Stacks in Benützung und welche für weitere Benützung frei sind. Jeder Versuch, etwas an eine bereits belegte Position im Stack zu speichern, resultiert in einem Ausnahmezustand. Wir werden diese Ausnahmezustandsbedingungen gleich besprechen, und auch ein Programm aufführen, das zeigt, wie wir das Problem des Stacküberlaufs lösen können.

Mittels eines Ladebefehls können wir Daten in den Stack des 8087 laden. Alle Ladebefehle transportieren die Daten dabei an die Spitze des Stacks. Befindet sich die Zahl im Speicher dabei noch nicht im temporären Gleitpunktformat, so wird diese Zahl vom 8087 als Teil der Ladeoperation in die 80-Bit Darstellung überführt. Umgekehrt nimmt ein Speicherbefehl einen Wert aus dem Stack des 8087 und legt ihn im Hauptspeicher ab. Wird auch in diesem Fall eine Datenkonversion benötigt, so führt sie der 8087 als Teil des Speicherbefehls aus. Der Transport eines Wertes aus dem Stack in den Speicher bewirkt aber nicht automatisch einen POP-Befehl auf den Stack des 8087. Einige Formen des Speicherbefehls lassen nämlich die Stackspitze für spätere Befehle unverändert.

Haben wir nun einmal Daten in den Stack des 8087 geladen, so können wir diesen Wert mit jeder beliebigen arithmetischen Anweisung verarbeiten. Der numerische Befehlssatz des 8087 erlaubt dabei Register/Register-Operationen ebenso wie Speicher/Register-Operationen. Ebenso wie beim 8088 muß einer der beiden arithmetischen Operanden immer aus einem Register kommen. Beim 8087 muß einer der Operanden immer das Spitzenelement des Stacks sein. Der andere Operand kann aus dem Speicher oder aus dem Registerstack kommen. Außerdem muß der Registerstack immer das Ziel einer arithmetischen Operation des 8087 sein. Der Arithmetikprozessor kann nämlich das Ergebnis einer arithmetischen Operation nicht direkt im Speicher ablegen. Ein getrennter Speicherbefehl (oder Speicher-und POP-Befehl) wird benötigt, um den Operanden nach Ausführung der Operation zurück in den Speicher zu transportieren. Einige arithmetische Befehle löschen außerdem das Spitzenelement aus dem Stack, ohne es im Speicher abzulegen.

Steuerwort

Der 8087 verfügt über zwei 16 Bit lange Kontrollregister. Das eine der beiden Register dient zur Steuerung des Inputs, das andere zur Statusangabe des Outputs. Diese Steuerregister erlauben es dem Programmierer, die Befehlsausführung des 8087 zu beeinflussen. Wir werden im weiteren nur auf einige der verschiedenen Möglichkeiten eingehen. Die Darstellung aller möglichen Steuerfunktionen wäre weit außerhalb des Darstellungsbereichs dieses Buches. Abbildung 7.7 zeigt den Aufbau der Steuerregister.

Das Steuerwort erlaubt es uns, zwischen zwei möglichen Behandlungsarten von unendlichen Zahlen zu unterscheiden. Die Standardmethode ist dabei der positive Abschluß des Zahlensystems. Dabei werden vom 8087 sowohl positive als auch negative unendliche Zahlen als eine einzige vorzeichenlose Unendlichkeit behandelt. Die andere Methode, der angepaßte Abschluß des Zahlensystems, unterschei-

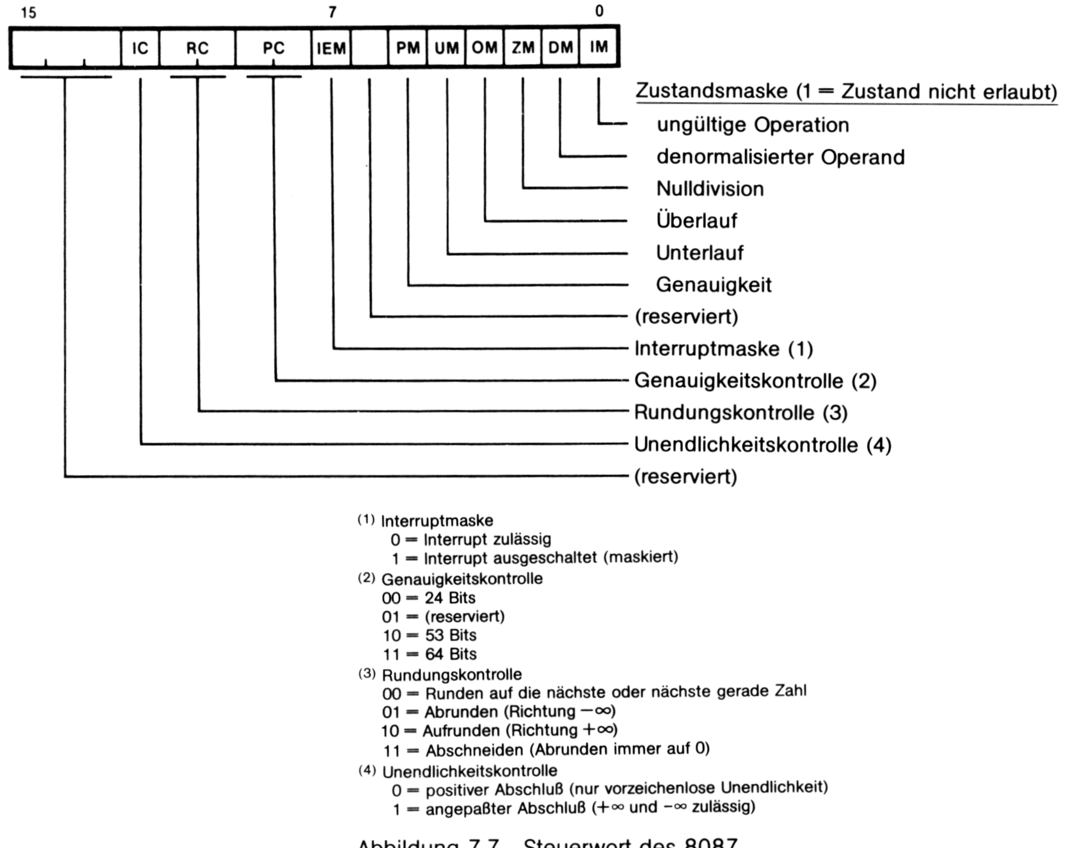


Abbildung 7.7 Steuerwort des 8087
(mit freundlicher Genehmigung von Intel; Coypright Intel 1980)

det zwischen positiven und negativen unendlichen Zahlen. Obwohl man annehmen könnte, daß der positive Abschluß des Zahlensystems zu einem Verlust von Informationen führen würde, ergeben sich dennoch keine irreführenden Resultate. Wir sollten deshalb den angepaßten Abschluß des Zahlensystems nur dann verwenden, wenn diese zusätzliche Information wirklich benötigt wird,und unser Programm darauf vorbereitet ist, mit diesen möglicherweise irreführenden Resultaten zurechtzukommen.

Der 8087 gestattet es uns außerdem, unter vier möglichen Arten des Rundens von Zahlen auszuwählen. Gerundet werden Zahlen dabei immer, wenn ein Ergebnis eine größere Genauigkeit erfordern würde, als es das Zahlensystem zuläßt. Die Methode des Rundens entscheidet dabei, welche Zahl als Ergebnis ausgewählt wird. Wir unterscheiden bei den einzelnen Rundungsmethoden zwischen dem Runden auf die nächstgrößere Zahl des Systems, dem Runden auf die nächstkleinere, dem Abrunden auf immer Null und dem Runden auf jeweils die nächste gerade Zahl. Der 8087 erlaubt es uns außerdem, die 64-Bit Genauigkeit des temporären Gleitpunktformats einzuschränken. Damit werden einige Forderungen des

vorgeschlagenen IEEE-Standards erfüllt. Diese Möglichkeit erhöht allerdings die Verarbeitungszeit und sollte nicht verwendet werden, nur um die Befehle des 8087 kompatibel mit einigen vielleicht vorexistierenden Programmen zu machen.

Der 8087 kann eine ganze Anzahl von Fehlern feststellen, die wir als Ausnahmezustände bezeichnen. Der 8087 kann zusätzlich Unterbrechungen erzeugen, um diese Ausnahmezustände zu signalisieren. Das Steuerwort verfügt dazu über eine Anzahl von Bits, die dem Programmierer die Entscheidung überlassen, welcher Ausnahmezustand nun einen Interrupt erzeugen soll und welcher vielleicht auf andere Weise behandelt wird. Diese Bits bezeichnen wir als Interruptmaske, da sie dazu verwendet werden können, das Auftreten eines Interrupts zu maskieren, also zu verhindern.

Es ist nicht nötig, daß jeder dieser einzelnen Ausnahmezustände einen Interrupt erzeugt. Der 8087 verfügt nämlich bereits intern über eine große Anzahl von Fehlerbehandlungsroutinen. Für jeden der möglichen Ausnahmezustände verfügt der 8087 über eine Standardlösung. So erzeugt beispielsweise eine Division durch 0 einen Interrupt, falls dieser zugelassen ist. Hat der Programmierer dagegen diesen Interrupt nicht zugelassen, so benützt der 8087 einen festgelegten unendlichen Wert als Ergebnis einer Nulldivision. Dieser Wert wird in darauffolgenden Rechenoperationen weiterverwendet, wobei dann beim Endergebnis mit angegeben ist, daß während der Verarbeitung ein bestimmter Fehler auftrat.

Dennoch könnte es zum Beispiel in Ihrem Programm notwendig sein, daß ein gerade auftretender Ausnahmezustand sofort bearbeitet wird. Es könnte auch sein, daß Sie Ihre eigenen Routinen zur Abhandlung dieser Sonderfälle einsetzen wollen. Dazu ist im IBM PC der Ausnahmeinterrupt des 8087 mit dem nichtmaskierbaren Interrupt (NMI) verbunden. Dies ist auch der gleiche Interrupt, der Parityfehler signalisiert. Benützen wir nun in einem Programm den Interruptmechanismus des 8087, so müssen wir den Interruptvektor für NMI so verändern, daß er auf eine Unterbrechungsbehandlungsroutine für den 8087 zeigt. Diese Routine muß nun auch alle evtl. auftretenden Parityfehler abhandeln. Im Kapitel 10 werden wir ein Beispiel dafür sehen, wie wir eine Interruptroutine durch unsere eigene ersetzen.

Betrachten wir als weiteres Beispiel für einen Ausnahmezustand den Stacküberlauf, den wir bereits vorher erwähnten. Schieben wir im Lauf eines Programmes einen neunten Eintrag in den Stack, so antwortet der 8087 mit einem Ausnahmeinterrupt. Ist dieser Interrupt nicht zugelassen, so kennzeichnet der 8087 den Befehl als ungültigen Befehl und legt als Ergebnis den Wert NAN (Not A Number, also ungültig) ab. Planen wir komplexe arithmetische Vorgänge, die mehr als acht Stackpositionen benötigen, so können wir diesen Ausnahmeinterrupt zu unseren Gunsten auswerten. Tritt nämlich ein Stacküberlauf ein, so kann die Interruptroutine einige der unteren Stackeinträge in den Speicher übertragen. Das Programm kann also diese Stackpositionen für weitere Benutzung freimachen. Es gibt im übrigen auch einen entsprechend Stackunterlauf-Ausnahmezustand. Diese Unterbrechung tritt dann auf, wenn wir eine leere Stackposition benutzen. Unsere Fehlerbehandlungsroutine kann auch diesen Zustand abhandeln. Sie kann beispielsweise den früheren Wert des Stacks dadurch herstellen, daß sie den Wert aus einem Speicherbereich zurückladt.

Wir werden später sehen, daß der 8087 über zusätzliche Statusinformationen verfügt, die es ermöglichen, diese Ausnahmezustände in einer Unterbrechungsroutine zu behandeln. Dabei versieht uns der 8087 mit vollständigen Informationen über den Befehl, der diesen Ausnahmezustand verursachte.

Statuswort

Das Statuswort des 8087 teilt uns den aktuellen Zustand des Coprozessors mit. Abbildung 7.8 zeigt den Aufbau des Statusworts. Das Statuswort enthält Bits für jeden Ausnahmezustand, so daß die Unterbrechungsroutine in jedem Fall den Grund der Störung erkennen kann. Das Statuswort verfügt außerdem über ein Bit, das angibt, ob der Prozessor gerade arbeitet oder nicht. Dieses Bit wird auch nach außen weitergegeben, um eine Synchronisation mit dem 8088 zu ermöglichen. Das Statuswort enthält zusätzlich den Pointer auf die aktuelle Stackspitze innerhalb der Registers des 8087.

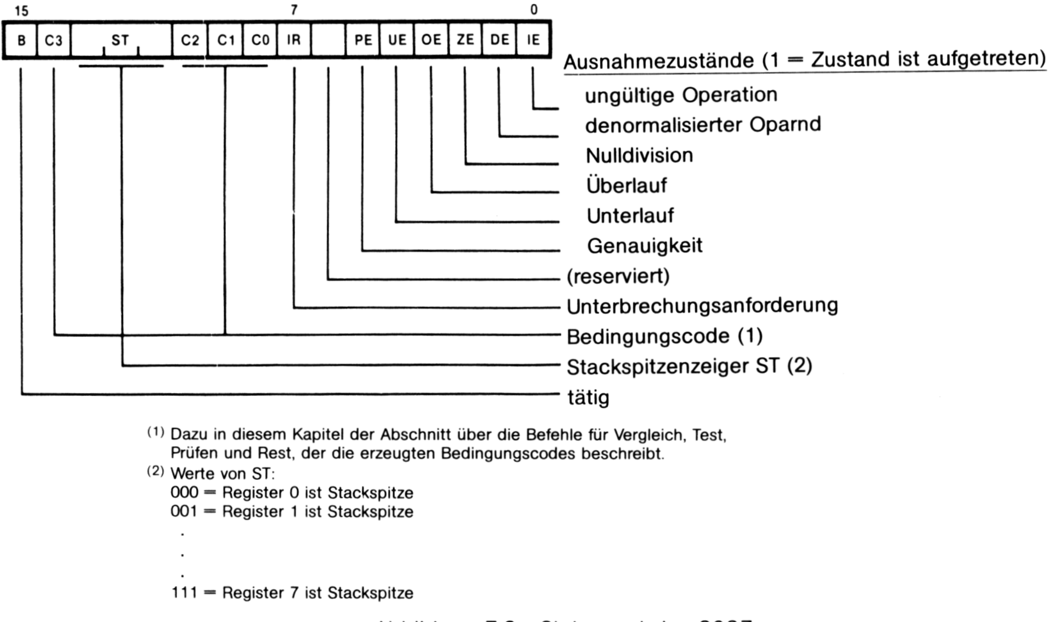


Abbildung 7.8 Statuswort des 8087
(mit freundlicher Genehmigung von Intel; Coypright Intel 1980)

Der vermutlich am meisten benützte Teil des Statusworts des 8087 ist das Bedingungscode-Register. Das Statuswort enthält dazu vier Bits, die von den Operationen des 8087 gesetzt werden. Zwei dieser Bedingungscode-Bits entsprechen dabei direkt dem Carry- und dem Nullflag des 8088. Sie stehen in der Tat sogar an denselben Bitpositionen innerhalb des höherwertigen Bytes des Statusworts des 8087. Diese Anordnung ist für uns dann von Vorteil, wenn wir das Statuswort im Speicher ablegen. Dazu transportieren wir das höherwertige Byte des Statusworts in das AH-Register. Der Befehl SAHF setzt dann Carry- und Nullflag in Abhängigkeit

von den Ergebnissen des Vergleichs im 8087. Da alle Zahlen innerhalb des 8087 vorzeichenbehaftete Gleitpunktzahlen sind, genügen die beiden Flags immer, um die Reihenfolge von zwei Zahlen festzustellen. Wir werden später noch mit einigen Beispielen arbeiten, die das Statuswort für den Vergleich von Zahlen verwenden. Die restlichen beiden Bits des Bedingungscode-Registers werden in Zusammenhang mit einem speziellen Befehl des 8087 verwendet, der es uns erlaubt, einen Test auf alle vom 8087 unterstützten besonderen Zahlen durchzuführen. Da viele dieser Zahlen spezielle Verarbeitungsweisen erfordern, verfügen wir mit dem Bedingungscode-Register über einen Mechanismus, sie herauszufiltern.

Befehlssatz des 8087

Wir müssen den 8087 ganz einfach als Erweiterung des 8088 betrachtet. Der 8087 erweitert dabei den verfügbaren Befehlssatz. Die Speicheradressierung wird analog zum 8088 durchgeführt. Wie wir bereits früher gesehen haben, geschieht dies dadurch, daß der 8088 die eigentlichen Speicheradressen erzeugt, während der 8087 die darin enthaltenen numerischen Operanden verarbeitet.

Der 8087 verfügt über einen Satz von acht Gleitpunktregistern, die als Stack angeordnet sind. Dabei wird die Spitze des Stack mit einem 3-Bit Pointer lokalisiert, der im Statuswort enthalten ist. Wenn wir nun das Statuswort lesen, so können wir sehen, welches der acht Register gerade die Spitze des Stacks darstellt. Wie wir zusätzlich sehen werden, benötigt ein Programm diese Information nur sehr selten.

Die Adressierung des Registerstacks des 8087 erfolgt immer relativ zur aktuellen Spitze des Stacks. Die Spitze des Stacks wird dabei als ST(0) oder ST bezeichnet und im Assembler auch so dargestellt. Das nächstfolgende Element des Stacks bezeichnen wir dann als ST(1). Das zweite Element wäre ST(2), usw., und das letzte Element schließlich ST(7). Aus Einfachhheitsgründen werden wir die Klammer bei der Bezeichnung der Stackelemente weglassen, so daß wir in Zukunft schreiben ST0, ST1, ST2, usw.

Sehen wir uns nun einmal an, wie wir in einem Programm die einzelnen Stackelemente ansprechen können. Der Befehl

```
FADD    ST0,ST3
```

addiert den Wert an der Stackspitze mit dem vierten Element des Stacks und speichert das Ergebnis wieder im obersten Element des Stacks. Abbildung 7.9(a) zeigt den Ablauf dieses Befehls. Das Ansprechen der Register ST0 und ST3 geschieht jeweils in direkter Abhängigkeit vom Inhalt des Stackpointers. Im Beispiel (a) addierten wir die Werte in A und D, wobei das Ergebnis A+D an die Stelle A zurückgespeichert wurde. In Abbildung 7.9(b) wurde ein weiteres Element E in den Stack vor Ausführung des gleichen Befehls geschoben. Auch hier wird wieder das Spitzenelement des Stacks mit dem vierten Element addiert. Im Beispiel (b) werden nun E und C addiert, wobei das Ergebnis E+C an die Stelle E gespeichert wird. Das Hinzufügen des Elements E an den Stack läßt also die Positionierung der Operanden innerhalb des 8087 unverändert, doch wurde ihre Position in Hinsicht auf die Stackspitze verändert. ST3 bleibt dabei immer das vierte Element des Stacks, ohne Rücksicht darauf, wohin der Stackpointer nun aktuell zeigt.

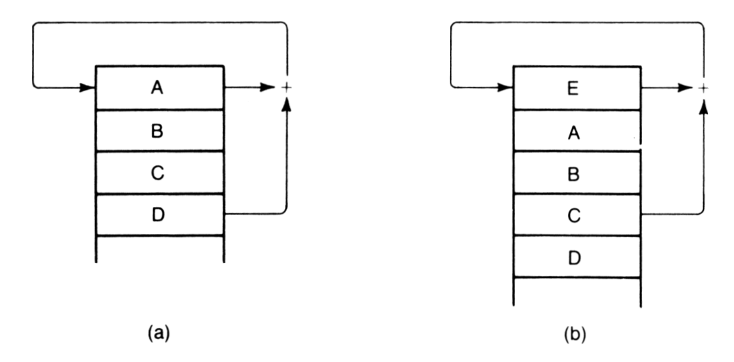


Abbildung 7.9 Ablauf des Befehls FADD ST0,ST3

Wir können den Befehlssatz des 8087 in drei große Kategorien aufteilen. Die erste Kategorie ist Datentransport – Laden und Speichern von Daten von und nach dem 8087. Die zweite Kategorie ist Prozessorsteuerung – Befehle, die den internen Ablauf des 8087 steuern. Die dritte Kategorie von Befehlen macht die eigentliche Stärke des 8087 aus, nämlich die arithmetischen Befehle. Jede der einzelnen Gruppen werden wir uns etwas genauer ansehen. Wir werden dabei nicht alle Anweisungen bis ins kleinste Detail besprechen. Doch können wir einen Großteil der besprochenen Befehle auch an Beispielen genau verdeutlichen. Es ginge nämlich über den Rahmen dieses Buches hinaus, alle Möglichkeiten der Anwendung der Befehle des 8087 genau darzustellen.

Befehle zum Datentransport

Die Gruppe der Datentransportbefehle für den 8087 besteht aus drei grundlegenden Anweisungen. Der Ladebefehl nimmt dabei Daten und schiebt sie in den Registerstack des 8087. Normalerweise kommen diese Daten aus dem Speicher, doch der Ladebefehl kann ebenso einen Wert aus dem Stack nehmen und ihn wiederum in den Stack speichern. Der Speicherbefehl nimmt die Daten aus der Spitze des Stacks und legt sie an beliebiger Stelle im Speicher ab. Der Tauschbefehl tauscht einfach zwei Zahlen innerhalb des Registerstacks des 8087 aus.

Abbildung 7.10 zeigt die assemblierten Befehle zum Datentransport. Dabei haben wir einen Satz von Makros für den 8087 durch die Anweisungen

```
IF1
INCLUDE 87MAC.LIB
ENDIF
```

in unser Programm eingefügt.

Diese Befehlsfolge bewirkt, daß während des ersten Durchlaufs des Assemblers, wo die Makrobearbeitung stattfinden muß, die Makros für den 8087 mitverwendet werden. Im zweiten Durchlauf des Assemblers werden diese Makros dann nicht mehr benötigt, da sie bereits durch die entsprechenden Befehle ersetzt sind. Der Befehl ENDIF ist dabei der einzige, der später in der Assemblerliste auftritt.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 7.10 8087 Data Transfer Instructions

1                                              PAGE    ,132
2                                              TITLE   Figure 7.10 8087 Data Transfer Instructions
3                                      ENDIF
4
5       0000                           CODE    SEGMENT
6                                              ASSUME  CS:CODE, DS:CODE
7
8       0000                           word_integer    label   word
9       0000                           short_integer   label   dword
10      0000                           long_integer    label   qword
11      0000                           bcd_integer     label   tbyte
12      0000                           short_real      label   dword
13      0000                           long_real       label   qword
14      0000                           temporary_real  label   tbyte
15
16                                             FILD    WORD_INTEGER
17      0000  9B                   +                 DB    09BH
18      0001  DF 06 0000 R         +                 ESC   38H,WORD_INTEGER
19                                             FILD    SHORT_INTEGER
20      0005  9B                   +                 DB    09BH
21      0006  DB 06 0000 R         +                 ESC   18H,SHORT_INTEGER
22                                             FILD    LONG_INTEGER
23      000A  9B                   +                 DB    09BH
24      000B  DF 2E 0000 R         +                 ESC   03DH,LONG_INTEGER
25                                             FBLD    BCD_INTEGER
26      000F  9B                   +           DB      09BH
27      0010  DF 26 0000 R         +           ESC     03CH,BCD_INTEGER
28                                             FLD     SHORT_REAL
29      0014  9B                   +                     DB    09BH
30      0015  D9 06 0000 R         +                     ESC   8,SHORT_REAL
31                                             FLD     LONG_REAL
32      0019  9B                   +                     DB    09BH
33      001A  DD 06 0000 R         +                     ESC   40,LONG_REAL
34                                             FLD     TEMPORARY_REAL
35      001E  9B                   +                     DB    09BH
36      001F  DB 2E 0000 R         +                     ESC   01DH,TEMPORARY_REAL
37
38                                             FLD     ST2
39      0023  9B D9 C2             +                     DB    09BH,0D9H,0C0H+ST2
40
41                                             FIST    WORD_INTEGER
42      0026  9B                   +                 DB    09BH
43      0027  DF 16 0000 R         +                 ESC   03AH,WORD_INTEGER
44                                             FIST    SHORT_INTEGER
45      002B  9B                   +                 DB    09BH
46      002C  DB 16 0000 R         +                 ESC   01AH,SHORT_INTEGER
47                                             FST     SHORT_REAL
48      0030  9B                   +                 DB    09BH
49      0031  D9 16 0000 R         +                 ESC   10,SHORT_REAL
50                                             FST     LONG_REAL
51      0035  9B                   +                 DB    09BH
52      0036  DD 16 0000 R         +                 ESC   02AH,LONG_REAL
53
54                                             FST     ST2
55      003A  9B DD D2             +                 DB    09BH,0DDH,0D0H+ST2
56
57                                             FISTP   WORD_INTEGER
58      003D  9B                   +                 DB    09BH
59      003E  DF 1E 0000 R         +                 ESC   03BH,WORD_INTEGER
60                                             FISTP   SHORT_INTEGER
61      0042  9B                   +                 DB    09BH
62      0043  DB 1E 0000 R         +                 ESC   01BH,SHORT_INTEGER
63                                             FISTP   LONG_INTEGER
64      0047  9B                   +                 DB    09BH
65      0048  DF 3E 0000 R         +                 ESC   03FH,LONG_INTEGER
66                                             FBSTP   BCD_INTEGER
67      004C  9B                   +           DB      09BH
68      004D  DF 36 0000 R         +           ESC     03EH,BCD_INTEGER
69                                             FSTP    SHORT_REAL
70      0051  9B                   +                 DB    09BH
71      0052  D9 1E 0000 R         +                 ESC   0BH,SHORT_REAL
72                                             FSTP    LONG_REAL
73      0056  9B                   +                 DB    09BH
74      0057  DD 1E 0000 R         +                 ESC   02BH,LONG_REAL
75                                             FSTP    TEMPORARY_REAL
76      005B  9B                   +                 DB    09BH
77      005C  DB 3E 0000 R         +                 ESC   01FH,TEMPORARY_REAL
78
79                                             FSTP    ST2
80      0060  9B DD DA             +                 DB    09BH,0DDH,0D8H+ST2
81
82                                             FXCH    ST2
83      0063  9B D9 CA             +                 DB    09BH,0D9H,0C8H+ST2
84
85                                             FLDZ
86      0066  9B D9 EE             +           DB      09BH,0D9H,0EEH
87                                             FLD1
88      0069  9B D9 E8             +           DB      09BH,0D9H,0E8H
89                                             FLDPI
90      006C  9B D9 EB             +           DB      09BH,0D9H,0EBH
91                                             FLDL2T
92      006F  9B D9 E9             +           DB      09BH,0D9H,0E9H
93                                             FLDL2E
94      0072  9B D9 EA             +           DB      09BH,0D9H,0EAH
```

```
95                                              FLDLG2
96      0075  9B D9 EC          +              DB    09BH,0D9H,0ECH
97                                              FLDLN2
98      0078  9B D9 ED          +              DB    09BH,0D9H,0EDH
99
100     007B                         CODE      ENDS
101                                             END
```

Abbildung 7.10 Datentransportbefehle des 8087

Der erste Befehl zum Datentransport, den wir besprechen werden, ist der Ladebefehl. Alle Befehle für den 8087 beginnen dabei mit einem großen F. So lautet der Ladebefehl für den 8087 beispielsweise FLD (Floating LoaD). Im Gegensatz zum 8088, bei dem der MOV-Befehl alle Formen von Daten transportieren konnte, gibt es hier verschiedene Befehle für die einzelnen Datentypen. Dies geschieht deshalb, da der Assembler zwar zwischen 4-Byte und 8-Byte Operanden unterscheiden kann, aber nicht, ob eine Zahl eine Gleitpunktzahl oder eine Integerzahl ist.

Handelt es sich beim Operanden um einen Integerwert, so verwenden wir den Befehl FILD. Damit können wir ein Wort (16 Bits), eine kurze Integerzahl (32 Bits), oder eine lange Integerzahl (64 Bits) laden. Zum Laden eines gepackten Dezimalwerts (80 Bits) benützen wir den Befehlscode FBLD. Das B steht dabei für BCD-Wert. Der Befehl FLD dient schließlich zum Laden von Gleitpunktzahlen. Der Assembler (und der Makrobefehl des 8087) finden dann heraus, welchen Typ von Integer- oder Gleitpunktzahl wir wirklich laden wollen.

Für alle Befehle des 8087, die sich auf den Speicher beziehen, gilt dabei folgende Konvention: F gefolgt von I für Integer, B für BCD-Zahlen und kein Zeichen für Gleitpunktzahlen. Wir werden im weiteren sehen, daß die Speicherbefehle nach den gleichen Konventionen arbeiten, wie es auch die arithmetischen Befehle tun, wenn sie sich auf Speicheroperanden beziehen.

In Abbildung 7.10 sehen wir einen Ladebefehl für jeden der sieben möglichen Datentypen des 8087. Der Ladebefehl bezieht sich dabei jedesmal auf einen Speicheroperanden. Der 8087 konvertiert die Daten dann von der externen Darstellung in das temporäre Gleitpunktformat. Die so konvertierte Zahl wird in den Stack geschoben, wobei sich die Größe des Stacks um 1 erhöht. Sollten wir dabei versuchen, einen Wert in den Stack zu schieben, wenn dieser bereits acht Werte enthält, so signalisiert uns der 8087 eine Ausnahmebedingung — Stacküberlauf. Solange wir nun nicht in unserem Programm eine eigene Routine zur Abhandlung dieses Sonderfalls vorgesehen haben, kennzeichnet die eingebaute Sonderfallbehandlung diesen Wert als unbestimmt. Dies bedeutet, daß alle weiteren Befehle, die auf diesen Wert zugreifen, unbestimmte Resultate erzeugen werden. Machen wir also einen Fehler, so sorgt der 8087 dafür, daß dieser Fehler nicht unbemerkt bleibt.

Die letzte Form des Ladebefehls schließlich nimmt ein Element aus dem Stack und schiebt es an die Stackspitze. So dupliziert beispielsweise der Befehl

```
FLD     ST0
```

die Spitze des Stacks. Die beiden ersten Stackelemente haben dann den gleichen Wert.

```
FLD     ST3
```

In diesem Fall schieben wir eine Kopie des vierten Elements des Stacks in diesen. Halten wir dabei fest, daß der Wert, der sich vorher an der Stelle ST3 befand, nun an der Stelle ST4 steht.

Sehen wir uns nun einmal den Maschinencode an, der von den Befehlen erzeugt wurde. Da in diesem Text die Anweisungen für den 8087 mit Makros erzeugt wurden, können wir auf einfache Weise feststellen, woher die einzelnen Werte kommen. Zunächst beginnt jeder Befehl mit dem Byte 09BH, was einen WAIT-Befehl darstellt. Wie wir uns erinnern, muß der 8087 mit dem 8088 synchronisiert werden. Würde nämlich der 8088 versuchen, einen weiteren Befehl für den 8087 anzusprechen, bevor der 8087 den vorausgegangenen beendet hat, so würde der 8087 falsche Ergebnisse produzieren. Aus diesem Grunde beinhalten praktisch alle Makros für den 8087 den WAIT-Befehl, um solchermaßen die Synchronisation sicherzustellen. (Die Befehle ohne Synchronisation sind Steuerbefehle, die normalerweise das letzte Ergebnis nicht benötigen. Wir können diese Befehle leicht erkennen, da sie alle mit FN beginnen, wobei das N bedeutet, daß keine Synchronisation nötig ist).

Im weiteren können wir außerdem erkennen, daß die Befehle des 8087 nur besondere Formen des ESC-Befehls sind. Zur Angabe der Speicheradresse beinhaltet der ESC-Befehl deswegen zwei Operanden. Der erste Operand legt dabei fest, um welchen ESC-Befehl es sich handelt, der zweite Operand bestimmt die Speicherstelle. Der ESC-Befehl kann dabei 2, 3, oder 4 Bytes lang sein, jeweils in Abhängigkeit von der Größe des Displacements, das durch das mod-r/m Byte bestimmt ist. Zusammen mit dem WAIT-Befehl ergibt sich so eine maximale Größe von fünf Bytes für einen 8087-Befehl.

Den Speicherbefehl gibt es in zwei Ausgaben. Bei der ersten Form des Befehls wird der Inhalt der Spitze des Stacks an eine gewünschte Speicherstelle übertragen. Als Teil der Befehlsausführung konvertiert der 8087 dabei die Daten vom temporären Gleitpunktformat in das gewünschte externe Format. Die Befehle hierzu sind FST und FIST. (Halten wir fest, daß die Konventionen zur Bezeichnung von Befehlen auch hier weiter eingehalten werden.) Wir können den Befehl außerdem dazu verwenden, den Inhalt der Stackspitze an eine beliebige Stelle innerhalb des Stacks zu speichern.

Wir müssen einschränkend bemerken, daß es der Befehl FST nicht gestattet, die Daten in Form aller möglichen Datentypen zu speichern. Nur die vier großen Datentypen sind zugelassen — lange und kurze Integerzahlen, lange und kurze Gleitpunktzahlen. Es werden also nicht alle externen Datentypen von diesem Befehl unterstützt, vermutlich deswegen, da die Entwickler des 8087 der Ansicht waren, dies sei nicht nötig. Und zu dieser Ansicht kamen sie wahrscheinlich wegen des nächsten Befehls.

Die zweite Ausführung des Speicherbefehls manipuliert nämlich zusätzlich zum Speichern der Daten auch den Stackpointer. Die Befehle FSTP (sowie FISTP und FBSTP) transportieren also ebenso Daten vom 8087 in den Speicher. Allerdings wird durch diese Befehle der Wert auch aus der Spitze des Stacks entnommen, ist also nachher nicht mehr verfügbar. Und diese Reihe von Befehlen unterstützt nun alle möglichen externen Datentypen. Die Entwickler des 8087 mußten nämlich in

gewisser Hinsicht mit den Befehlen sparsam umgehen, weshalb nur die Befehle FLD und FSTP alle möglichen externen Datenstrukturen unterstützen. Sämtliche übrigen Befehle, die sich auf den Speicher beziehen, können nur die vier großen Datentypen bearbeiten. Diese vier großen Datentypen sind allerdings auch die wichtigsten, so daß die Verwendung der anderen Formate auf die Befehle FLD und FSTP eingeschränkt werden konnte.

Der nächste Befehl zum Datentransport ist der Befehl FXCH. Er dient zum Vertauschen von Daten und tauscht dabei den Inhalt der Stackspitze mit einem beliebigen Register innerhalb des Stacks aus. Halten wir dabei fest, daß wir bei diesem Befehl nur ein anderes Element des Stacks selbst als Operand verwenden können. Wir können also nicht mit einem einzelnen Befehl die Spitze des Stacks mit einer beliebigen Speicherstelle vertauschen. Dazu würden wir mehrere Befehle benötigen, und außerdem einen zusätzlichen Zwischenspeicher. Im Gegensatz zum 8088 kann der 8087 nämlich jeweils nur einen Lese- oder Schreibzyklus innerhalb eines Befehls durchführen, jedoch nicht beide zugleich.

Befehl	Konstante
FLDZ	0
FLD1	1
FLDPI	PI
FLDL2T	LOG2(10)
FLDL2E	LOG2(e)
FLDLG2	LOG10(2)
FLDLN2	LOGe(2)

Abbildung 7.11 Konstanten des 8087

Mit den restlichen Befehlen innerhalb der Gruppe der Datentransportbefehle behandeln wir numerische Konstanten. Wir laden mit diesen Befehlen also vordefinierte Konstanten an die Spitze des Stacks. Dabei können wir alle in einem Programm notwendigen numerischen Konstanten darstellen. Die Werte wurden außerdem so gewählt, daß sie die Verarbeitung von transzendenten und trigonometrischen Funktionen erleichtern. Wir werden einige dieser Konstanten in unserem Beispielprogramm verwenden. Die Tabelle in Abbildung 7.11 zeigt dabei den jeweils in das ST0-Register geladenen Wert. Der Befehlscode wurde so gewählt, daß er aussagefähig hinsichtlich des jeweils zu ladenden Werts ist.

Steuerbefehle

Die Steuerbefehle für den 8087 haben nichts mit Arithmetik zu tun. Sie sind jedoch nötig, um den Ablauf innerhalb des 8087 zu steuern. In Abbildung 7.12 sehen wir die Steueranweisungen für den 8087.

Viele dieser Steuerbefehle können ausgeführt werden, ohne auf den Abschluß des vorausgehenden 8087-Befehls zu warten. In Abbildung 7.12 wurden diese Befehle so assembliert, daß sie WAIT-Befehle mit beinhalten. Der Befehlscode im Kommentarfeld zeigt dagegen den Befehl ohne WAIT. Der Assembler kennzeichnet die Befehle ohne WAIT durch die Angabe FN in den ersten beiden Buchstaben des Befehlscodes.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 7.12  8087 Control Instructions

1                                              PAGE    ,132
2                                              TITLE   Figure 7.12  8087 Control Instructions
3                                      ENDIF
4       0000                           CODE    SEGMENT
5                                              ASSUME  CS:CODE
6
7       0000                           status_word     label   word
8       0000                           control_word    label   word
9       0000                           environment     label   byte    ; 14 byte area
10      0000                           state           label   byte    ; 94 byte area
11
12                                             FINIT                   ; FNINIT
13      0000  9B DB E3             +           DB     09BH,0DBH,0E3H
14                                             FENI                    ; FNENI
15      0003  9B DB E0             +           DB     09BH,0DBH,0E0H
16                                             FDISI                   ; FNDISI
17      0006  9B DB E1             +           DB     09BH,0DBH,0E1H
18                                             FLDCW   control_word
19      0009  9B                   +           DB     09BH
20      000A  2E: D9 2E 0000 R     +           ESC    0DH,control_word
21                                             FSTCW   control_word    ; FNSTCW
22      000F  9B                   +           DB     09BH
23      0010  2E: D9 3E 0000 R     +           ESC    0FH,control_word
24                                             FCLEX                   ; FNCLEX
25      0015  9B DB E2             +           DB     09BH,0DBH,0E2H
26                                             FSTENV  environment     ; FNSTENV
27      0018  9B                   +           DB     09BH
28      0019  2E: D9 36 0000 R     +           ESC    0EH,environment
29                                             FLDENV  environment
30      001E  9B                   +           DB     09BH
31      001F  2E: D9 26 0000 R     +           ESC    0CH,environment
32                                             FSAVE   state           ; FNSAVE
33      0024  9B                   +           DB     09BH
34      0025  2E: DD 36 0000 R     +           ESC    02EH,state
35                                             FRSTOR  state
36      002A  9B                   +           DB     09BH
37      002B  2E: DD 26 0000 R     +           ESC    02CH,state
38                                             FINCSTP
39      0030  9B D9 F7             +           DB     09BH,0D9H,0F7H
40                                             FDECSTP
41      0033  9B D9 F6             +           DB     09BH,0D9H,0F6H
42                                             FFREE   ST2
43      0036  9B DD C2             +           DB     09BH,0DDH,0C0H+ST2
44                                             FNOP
45      0039  D9 D0                +           DB     0D9H,0D0H
46                                             FWAIT
47      003B  9B                   +           DB     09BH
48      003C                            CODE   ENDS
49                                             END
```

Abbildung 7.12 Steuerbefehle für den 8087

In Abbildung 7.13 sind die Steuerbefehle des 8087 noch einmal zusammengefaßt.

Befehl	Aktion
FINIT	Initialisieren des 8087; Software-Reset
FENI	Unterbrechung für Ausnahmezustände erlauben
FDISI	Unterbrechung für Ausnahmezustände verhindern
FLDCW	Laden 8087 Steuerwort aus dem Speicher
FSTCW	Ablegen 8087 Steuerwort in den Speicher
FSTSW	Ablegen 8087 Statuswort in den Speicher
FCLEX	Löschen Ausnahmezustandsbits
FSTENV	Ablegen der 8087 Umgebung in den Speicher
FLDENV	Laden der 8087 Umgebung aus dem Speicher
FSAVE	Sichern des 8087 Zustands in den Speicher
FRSTOR	Laden des 8087 Zustands aus dem Speicher
FINCSTP	Erhöhen Stackpointer
FDECSTP	Erniedrigen Stackpointer
FFREE	Freigeben Register im Stack
FNOP	Nulloperation
FWAIT	entspricht WAIT-Befehl des 8088

Abbildung 7.13 Steuerfunktionen
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

Die Befehle FENI, FDISI und FCLEX behandeln alle die Ausnahmezustände des 8087. Das Steuerregister enthält dazu eine Interruptmaske, die auswählt, welche Ausnahmezustände einen Interrupt hervorrufen können. Die Befehle FENI und FDISI steuern ein allgemeines Interruptmaskenregister für den 8087. Sie ähneln dabei den Befehlen STI und CLI des 8088, mit der Ausnahme, daß sie nur die Interrupts des 8087 steuern. Der Befehl FCLEX löscht die Ausnahmezustandsbits des Statusregisters. Der 8087 zeichnet nämlich alle Ausnahmezustände auf, so daß, wenn innerhalb einer Befehlsfolge mehr als ein Fehler auftritt, alle diese Fehler im Statusregister wiedergespiegelt werden. Der Befehl FCLEX ist der einzige Weg, um diese Flags wieder zu löschen.

Die Bedeutung von Steuer- und Statuswort des 8088 haben wir bereits besprochen. Die Befehle FLDCW, FSTCW und FSTSW dienen zum Laden und Speichern dieser Register.

Die sogenannte Umgebung des 8087 enthält alle Register mit Ausnahme des Stacks. Diese Umgebung umfaßt 14 Datenbytes. Abbildung 7.14 zeigt den Aufbau der Umgebung, wenn der 8087 sie im Speicher ablegt. Das Speichern der Umgebung ist ein vernünftiger Weg, Ausnahmebedingungen des 8087 abzuarbeiten. Die Umgebung enthält nämlich alle Daten über den Ausnahmezustand. Eine 20-Bit Adresse beinhaltet dabei den letzten vom 8087 ausgeführten Befehl. Eine zweite Adresse zeigt auf die zuletzt in Anspruch genommenen Daten. Außerdem befindet sich der letzte vom 8087 ausgeführte Befehl in dieser Umgebung.

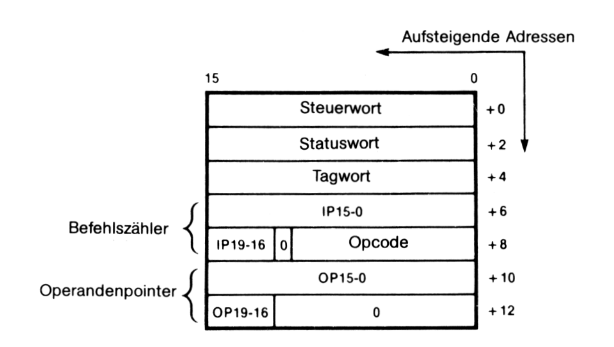


Abbildung 7.14 Umgebung des 8087
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

Der Status des 8087 setzt sich zusammen aus der Umgebung und den Datenregistern. Da es beim 8087 insgesamt acht Register von jeweils 10 Bytes Länge gibt, enthält der Status 94 Bytes. In Abbildung 7.15 sehen wir den Status des 8087, wie er im Speicher abgelegt wird. Dabei entspricht der Status der Umgebung mit dem Stack an ihrem Ende. In einem Programm können wir beispielsweise den Status des 8087 sicherstellen, wenn wir eine andere Aufgabe ausführen wollen oder wenn eine Unterbrechungsroutine die Benutzung des 8087 erfordert. Wir können den Status wieder zurückspeichern, wenn die vorausgegangene Routine wieder die Kontrolle erhält.

Zwei Befehle zur Manipulation des Stackpointers erfüllen genau diesen Zweck. Sie verändern nämlich nur den Stackpointer. Alle Daten in den Registern bleiben so wie sie sind; das bedeutet, auch das Statuswort wird nicht modifiziert. Ein Erhöhen des Stackpointers entspricht also nicht einem Lesen von Daten aus dem Stack, denn der Statuswert für diesen Speicherbereich zeigt immer noch an, daß sich Daten im Register befinden. Sollten wir nun weitere Daten hinzuladen wollen, wäre das Ergebnis ein Stacküberlauf. Zum direkten Freigeben einer Speicherstelle des Stacks dient dagegen der Befehl FFREE, der den entsprechenden Statuswert setzt, um anzuzeigen, daß sich dort keine gültigen Daten mehr befinden. Doch der Befehl FFREE verändert den Stackpointer nicht. Wollen wir nun ganz einfach die Daten, die sich an der Spitze des Stacks befinden, wegwerfen, werden wir dazu einen arithmetischen Befehl verwenden, der diesen Zweck ausreichend erfüllt, den wir aber erst später besprechen werden.

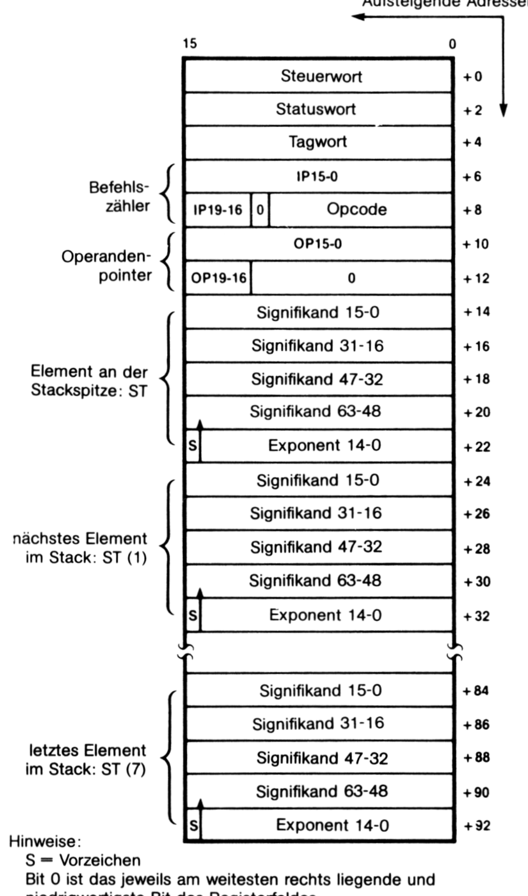


Abbildung 7.15 Status des 8087
(mit freundlicher Genehmigung von Intel; Copyrigth Intel 1980)

Arithmetische Befehle

Die arithmetischen Befehle sind das eigentliche Herz des 8087. Der 8087 bestichtdabei durch eine schnelle, hochpräzise Ausführung von numerischen Operationen. Der 8087 verfügt nicht nur über die allgemein üblichen vier Grundrechnungsarten wie Addition, Subtraktion, Multiplikation und Division, sondern auch über einen großen Bereich von transzendenten und trigonometrischen Funktionen, die notwendig sind, um den gesamten Bereich der numerischen Datenverarbeitung abzudecken.

Abbildung 7.16 zeigt einige der assemblierten Befehle für die vier Grundrechnungsarten. Dabei zeigen wir nur den Befehl FADD in allen seinen möglichen Variationen. Bevor wir uns die einzelnen Anweisungen ansehen, untersuchen wir zunächst die Möglichkeiten der Verarbeitung. Abbildung 7.17 zeigt die Variationen.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 7.16  Arithmetic Instructions of 8087

1                                          PAGE    ,132
2                                          TITLE   Figure 7.16  Arithmetic Instructions of 8087
3                                  ENDIF
4       0000                       CODE    SEGMENT
5                                          ASSUME  CS:CODE,DS:CODE
6       0000                       word_integer    label   word
7       0000                       short_integer   label   dword
8       0000                       short_real      label   dword
9       0000                       long_real       label   qword
10
11                                         FADD
12      0000  9B D8 C1          +              DB     09BH,0D8H,0C1H
13                                         FADD    ST2
14      0003  9B D8 C2          +                      DB      09BH,0D8H,0C0H+ST2
15                                         FADD    ST0,ST2
16      0006  9B D8 C2          +                      DB     09BH,0D8H,0C0H+ST2
17                                         FADD    ST2,ST0
18      0009  9B DC C2          +                      DB     09BH,0DCH,0C0H+ST2
19                                         FIADD   word_integer
20      000C  9B                +              DB     09BH
21      000D  DE 06 0000 R      +              ESC    030H,word_integer
22                                         FIADD   short_integer
23      0011  9B                +              DB     09BH
24      0012  DA 06 0000 R      +              ESC    010H,short_integer
25                                         FADD    short_real
26      0016  9B                +                      DB     09BH
27      0017  D8 06 0000 R      +                      ESC    0,short_real
28                                         FADD    long_real
29      001B  9B                +                      DB     09BH
30      001C  DC 06 0000 R      +                      ESC    32,long_real
31                                         FADDP   ST2,ST0
32      0020  9B DE C2          +          DB      09BH,0DEH,0C0H+ST2
33
34                                         FSUB    ST2
35      0023  9B D8 E2          +                      DB     09BH,0D8H,0E0H+ST2
36                                         FISUB   word_integer
37      0026  9B                +              DB     09BH
38      0027  DE 26 0000 R      +              ESC    034H,word_integer
39                                         FSUBP   ST2,ST0
40      002B  9B DE E2          +              DB     09BH,0DEH,0E0H+ST2
41
42                                         FSUBR   ST2,ST0
43      002E  9B DC EA          +                      DB     09BH,0DCH,0E8H+ST2
44                                         FISUBR  short_integer
45      0031  9B                +              DB     09BH
46      0032  DA 2E 0000 R      +              ESC    015H,short_integer
47                                         FSUBRP  ST2,ST0
48      0036  9B DE EA          +          DB      09BH,0DEH,0E8H+ST2
49
50                                         FMUL    short_real
51      0039  9B                +                      DB     09BH
52      003A  D8 0E 0000 R      +                      ESC    1,short_real
53                                         FIMUL   word_integer
54      003E  9B                +              DB     09BH
55      003F  DE 0E 0000 R      +              ESC    031H,word_integer
56                                         FMULP   ST2,ST0
57      0043  9B DE CA          +          DB      09BH,0DEH,0C8H+ST2
58
59                                         FDIV    ST0,ST2
60      0046  9B D8 F2          +                      DB     09BH,0D8H,0F0H+ST2
61                                         FIDIV   short_integer
62      0049  9B                +              DB     09BH
63      004A  DA 36 0000 R      +              ESC    016H,short_integer
64                                         FDIVP   ST2,ST0
65      004E  9B DE F2          +          DB      09BH,0DEH,0F0H+ST2
66
67                                         FDIVR   ST2
```

```
68      0051  9B D8 FA         +                DB     09BH,0D8H,0F8H+ST2
69                                    FIDIVR  word_integer
70      0054  9B               +          DB    09BH
71      0055  DE 3E 0000 R     +          ESC   037H,word_integer
72                                    FDIVRP  ST2,ST0
73      0059  9B DE FA         +      DB     09BH,0DEH,0F8H+ST2
74
75      005C                    CODE  ENDS
76                                    END
Open conditionals: 13
```

Abbildung 7.16 Arithmetische Befehle des 8087

Wie wir in Abbildung 7.17(a) sehen, gibt es fünf verschiedene Methoden der Ausführung einer arithmetischen Anweisung. Die verwendeten Daten für jede einzelne arithmetische Anweisung können wir den fünf einzelnen Fällen in Beispiel (a) entnehmen. Im Fall 1 sehen wir nur den Operationscode selbst. In diesem Falle werden die Spitze des Stacks und das Stackregister ST1 verarbeitet, wobei das Ergebnis die Spitze des Stacks ersetzt. In Abbildung 7.17(a) sehen wir anhand des Additionsbefehls ein Beispiel für jeden möglichen Fall. Das Bild zeigt außerdem ein Schema für die Anwendung dieses Befehls.

	Befehlsform		Beispiel FADD		Aktion
1.	Fop		FADD		ST0←ST0+ST1
2.	Fop	STi	FADD	ST2	ST0←ST0+STi
	Fop	ST0,STi	FADD	ST0,ST2	ST0←ST0+STi
	Fop	STi,ST0	FADD	ST2,ST0	ST2←ST0+ST2
3.	FopP	STi,ST0	FADDP	ST2,ST0	ST2←ST0+ST2, Pop stack
4.	Fop	real-mem	FADD	REAL	ST0←ST0+REAL
5.	Fop	integer-mem	FIADD	INTEGER	ST0←ST0+INTEGER

(a)

Operation	Aktion
ADD	Ziel ← Ziel + Quelle
SUB	Ziel ← Ziel − Quelle
SUBR	Ziel ← Quelle − Ziel
MUL	Ziel ← Ziel × Quelle
DIV	Ziel ← Ziel / Quelle
DIVR	Ziel ← Quelle / Ziel

(b)

Abbildung 7.17 Arithmetische Befehle.
(a) Kombinationsmöglichkeiten der Datentypen; (b) arithmetische Befehle des 8087
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

In Fall 2 sehen wir eine Rechenoperation mit zwei Registern aus dem Stack des 8087. Eines der beiden Register muß dabei das Register an der Stackspitze sein. Ist die Stackspitze das Ziel, so kann diese Angabe entfallen, wobei wir dann nur noch das Quellregister angeben müssen. Wird dagegen ein anderes Register als Ziel verwendet, so müssen sowohl Ziel- als auch Quellregister spezifiziert werden.

In Fall 3 wird eine Stackoperation durchgeführt. Die Rechenoperation wird also mit zwei Operanden aus dem Stack durchgeführt, und daraufhin die Spitze des Stacks

gelöscht. Da in diesem Fall die Spitze des Stacks verschoben wird, kann diese auch nicht das Ziel für den Befehl sein. Ist das Ziel aber ST1, so erhalten wir wieder eine klassische Stackoperation. Hier werden die beiden gewünschten Elemente aus dem Stack entnommen, in der verlangten Form kombiniert und das Ergebnis wieder zurück in den Stack gespeichert. Natürlich kann bei dieser Operation jedes der beiden Register als Zielregister spezifiziert sein.

Die beiden restlichen Fälle behandeln Speicheroperanden. In Fall 4 ist der Speicheroperand eine kurze oder lange Gleitpunktzahl. Und in Fall 5 ist der Operand eine kurze oder Wortintegerzahl. So wie bei den Lade- und Speicherbefehlen zeigt das I auch hier an, daß der Speicheroperand eine Integerzahl ist.

Wenn wir uns noch einmal die Assemblerliste in Abbildung 7.16 ansehen, so bemerken wir, daß für den Befehl FADD nur vier verschiedene Speicheroperanden Verwendung finden. Zwei davon sind Integerzahlen — Wort-Integer- und kurze Integerzahlen, und zwei davon sind Gleitpunktzahlen — kurz und lang. Die arithmetischen Befehle können nicht direkt BCD-Zahlen, lange Integerzahlen oder temporäre Gleitpunktzahlen verarbeiten. In einem Programm müssen wir diese Werte also in ein Register laden, bevor wir mit ihnen zu rechnen beginnen.

In Abbildung 7.17(b) sehen wir die sechs möglichen arithmetischen Operationen. Halten wir dabei fest, daß der 8087 die vier möglichen Standardfunktionen erweitert, indem er für Subtraktion und Division eine Umkehrfunktion vorsieht. Addition und Multiplikation benötigen dabei keine Umkehrfunktion, da ihre Faktoren austauschbar sind. Die Anordnung der Operanden bei Subtraktion und Division ist dagegen kritisch. Es kann nämlich manchmal vorkommen, daß der Wert, der sich gerade im Quellregister befindet, nicht der Wert ist, den wir vom Zielregister subtrahieren wollen. In diesem Fall kann die Umkehrfunktion diese Aufgabe übernehmen.

Vergleichsbefehle

So wie der 8088 verfügt auch der 8087 über Befehle, die zwei Zahlen miteinander vergleichen. Der 8087 vernachlässigt dabei das Ergebnis des Vergleichs, setzt aber die Statusflags entsprechend dem Ergebnis. Wir müssen deshalb mit einem Programm zuerst das Statuswort im Speicher hinterlegen, bevor wir die Statusflags untersuchen können. Der einfachste Weg hierzu ist, die Statusflags in das AH-Register und dann in die Flags des 8088 weiterzuspeichern, um sie von diesem dann testen zu lassen.

In Abbildung 7.18 sehen wir die Assemblerliste für einen Vergleichsbefehl auf dem 8087. Da bei einem Vergleich immer die Spitze des Stacks mitverwendet wird, muß unser Programm nur ein Register oder einen Speicheroperanden spezifizieren. Nach dem Vergleich enthält das Statuswort eine Angabe über die Reihenfolge unserer beiden Zahlen. Die Tabelle in Abbildung 7.19 zeigt diesen Zusammenhang. Dabei sind nur zwei der Statusbits nötig, um das Ergebnis des Vergleichs wiederzuspiegeln. Sehen wir uns dabei in Abbildung 7.8 die Stellen C3 und C0 innerhalb des Statusworts an.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 7.18  Comparison Instructions of 8087

1                                        PAGE    ,132
2                                        TITLE   Figure 7.18  Comparison Instructions of 8087
3                                ENDIF
4        0000                    CODE    SEGMENT
5                                        ASSUME  CS:CODE,DS:CODE
6        0000                    word_integer    label   word
7        0000                    short_integer   label   dword
8        0000                    short_real      label   dword
9        0000                    long_real       label   qword
10
11                                       FCOM
12       0000  9B D8 D1          +               DB      09BH,0D8H,0D1H
13                                       FCOM    ST2
14       0003  9B D8 D2          +                       DB      09BH,0D8H,0D0H+ST2
15                                       FICOM   word_integer
16       0006  9B                +               DB      09BH
17       0007  DE 16 0000 R      +               ESC     032H,word_integer
18                                       FCOM    short_real
19       000B  9B                +                       DB      09BH
20       000C  D8 16 0000 R      +                       ESC     2,short_real
21
22                                       FCOMP
23       0010  9B D8 D9          +               DB      09BH,0D8H,0D9H
24                                       FICOMP  short_integer
25       0013  9B                +               DB      09BH
26       0014  DA 1E 0000 R      +               ESC     013H,short_integer
27                                       FCOMP   long_real
28       0018  9B                +                       DB      09BH
29       0019  DC 1E 0000 R      +                       ESC     35,long_real
30
31                                       FCOMPP
32       001D  9B DE D9          +       DB      09BH,0DEH,0D9H
33
34                                       FTST
35       0020  9B D9 E4          +       DB      09BH,0D9H,0E4H
36                                       FXAM
37       0023  9B D9 E5          +       DB      09BH,0D9H,0E5H
38
39       0026                    CODE    ENDS
40                                       END
Open conditionals: 2
```

Abbildung 7.18 Vergleichsbefehle des 8087

Das Quellprogramm in Abbildung 7.20 zeigt einen Programmausschnitt, in dem die Spitze des Stacks mit einem Wort im Speicher verglichen wird. Danach wird in Abhängigkeit vom Ergebnis des Vergleiches verzweigt. Halten wir dabei fest, daß in einem der vier möglichen Fälle die Operanden nicht miteinander verglichen werden konnten. Dieser Fall tritt dann ein, wenn eine der Zahlen NAN darstellte (Not A Number), oder aber eine der beiden Formen von Unendlichkeit. Wiederum sehen wir, daß der 8087 die Statusflags C3 und C0 an genau den richtigen Stellen ablegt. Sie entsprechen nämlich in ihrer Position direkt dem Null- (C3) und Carryflag (C0) des 8087. Wir können nun das höherwertige Byte des Statusworts des 8087 in das Flagregister des 8088 speichern und dann nach einem Testen dieser Flags einen bedingten Sprung ausführen. Dazu müssen nicht einmal die einzelnen Bits des Statusworts maskiert oder getestet werden.

C3	C0	Folge
0	0	ST > Quelle
0	1	ST < Quelle
1	0	ST = Quelle
1	1	ST und Quelle können nicht verglichen werden

Abbildung 7.19 Vergleichsfolge
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

Übrigens zeigen wir hier die Beispielprogramme für den 8087 nur noch in Quellform, und nicht mehr als Assemblerliste. Wir tun dies, da die Makroexpansion die Liste unnötig aufbläht, was es uns schwer machen würde, der Logik des Progamms zu folgen. Ist allerdings aus der Assemblerliste eines Programms etwas Wichtiges zu erkennen, werden wir selbstverständlich wieder diese Liste anstelle des Quellprogramms anführen.

Kehren wir zurück zur Assemblerliste in Abbildung 7.18. Dort sehen wir noch einige zusätzliche Variationen des Befehls FCOM. Natürlich gibt es hier auch die Integerversion, FICOM. Die Befehle FCOMP und FICOMP sind dabei identisch zum Befehl FICOM, mit der Ausnahme, daß der 8087 nach jedem Befehl den Stack um 1 erniedrigt. Dies ermöglicht es uns, mit dem 8087 zwei Zahlen miteinander zu vergleichen, ohne uns darum kümmern zu müssen, daß wir den ersten Operanden eigentlich nach dem Vergleich aus dem Stack löschen müßten.

```
        PAGE    ,132
        TITLE   Figure 7.20  Comparison branching
IF1
INCLUDE 87MAC.LIB
ENDIF
        CODE    SEGMENT
        ASSUME  CS:CODE, DS:CODE

WORD_INTEGER    LABEL   WORD
STATUS_WORD     DW      ?

;----- This code compares the stack top to word_integer

        FICOM   WORD_INTEGER    ; Compare word to ST
        FSTSW   STATUS_WORD     ; Store the status word
        FWAIT                   ; Wait for store complete
        MOV     AH,BYTE PTR STATUS_WORD+1       ; Get into AH
        SAHF                    ; Move to flags [C0=CF, C3=ZF]
        JB      CONTINUE        ; Test C0 flag, jump if zero
        JNE     ST_GREATER      ; Test C3 flag
ST_EQUAL:                       ; Here if C3=1, C0=0
;       . . .
ST_GREATER:                     ; Here if C3=0, C0=0
;       . . .
CONTINUE:
        JNE     ST_LESS         ; Test C3 flag
UNORDERED:                      ; Here if C3=1, C0=1
;       . . .
ST_LESS:                        ; Here if C3=0, C0=1
;       . . .

CODE    ENDS
        END
```

Abbildung 7.20 Vergleich und Sprung

Beim Befehl FCOMPP gibt es keine vom Benutzer spezifizierten Operanden. Er vergleicht immer die beiden ersten Elemente des Stacks. Nach Ausführung des Vergleichs werden beide Elemente aus dem Stack entfernt.

Diese Kombination aus Vergleich- und POP-Befehl bietet einen möglichen Weg, Werte aus dem Stack zu entfernen. Da nämlich der 8087 über keine eigenen Befehle verfügt, um Operanden aus dem Stack zu entfernen, können wir dazu diesen Befehl verwenden. Natürlich wird dadurch auch das Statusregister verändert. Wir können die Befehle also nicht verwenden, wenn wir auf den Inhalt der Statusbits angewiesen sind. In den meisten Fällen bieten diese Befehle jedoch einen schnellen Weg, entweder einen oder zwei Operanden aus dem Stack zu entfernen. Da der 8087 jedesmal eine Fehlermeldung abgibt, wenn der Stack überläuft, müssten wir nach jeder beendeten Rechenoperation alle verwendeten Operanden aus dem Stack entfernen.

Es gibt noch zwei weitere spezielle Vergleichsbefehle. Der erste, FTST, ist eine Überprüfung des obersten Stackelements auf Null. Dieser Befehl erlaubt es uns, auf einfache und schnelle Weise das Vorzeichen des obersten Stackeintrags festzustellen. Verwenden wir dazu noch einmal die Tabelle in Abbildung 7.19. Sie zeigt die beiden Statusbits nach Ausführung eines Vergleichs. Um die Ergebnisse des Vergleichs auf Null festzustellen, müssen wir nur den Operanden „Source" durch Null ersetzen.

Der Befehl FXAM ist genaugenommen kein Vergleichsbefehl. Obwohl er nämlich mit der Spitze des Stacks arbeitet, wird diese Stackspitze mit keinem anderen Wert verglichen. Dagegen setzt der Befehl FXAM die vier Bits des Statusregisters (C3 bis C0), um anzuzeigen, welche Art von Zahlen sich an der Spitze des Stacks befinden. Da der 8087 nämlich neben den normalisierten Zahlen noch eine ganze Reihe von anderen Zahlenarten verarbeiten kann, teilt Ihnen der Befehl FXAM mit, welche Zahl sich aktuell an der Spitze des Stacks befindet. Im Abbildung 7.21 sehen wir die Kombination der Statusbits für jede mögliche Zahl.

Wenn Ihr Rechenprogramm nichts besonders Verrücktes macht oder die Grenzen der Datendarstellung des 8087 sprengt, werden Sie wahrscheinlich diese Zustände des FXAM-Befehles nie zu Gesicht bekommen. Als normale Daten betrachten wir dabei positive oder negative Zahlen oder die Null. Der unbestimmte Zustand des obersten Stackregisters sollte dabei auf eine Fehlerbedingung beschränkt bleiben. Ein Unterprogramm könnte beispielsweise diesen Test durchführen, um zu überprüfen, welcher Parameter an der Spitze des Stacks übergeben wurde.

C3	C2	C1	C0	Bedeutung	C3	C2	C1	C0	Bedeutung
0	0	0	0	+ unnormal	1	0	0	0	+0
0	0	0	1	+ NAN	1	0	0	1	leer
0	0	1	0	- unnormal	1	0	1	0	-0
0	0	1	1	- NAN	1	0	1	1	leer
0	1	0	0	+ normal	1	1	0	0	+,denormal
0	1	0	1	+ unendlich	1	1	0	1	leer
0	1	1	0	- normal	1	1	1	0	- denormal
0	1	1	1	- unendlich	1	1	1	1	leer

Abbildung 7.21 Bedingungscode nach FXAM
(mit freundlicher Genehmigung von Intel; Copyright Intel 1980)

Die restlichen Zahlenwerte sind Antworten des 8087 auf eine Fehlerbedingung. Sobald der 8087 nämlich eine Fehlerbedingung entdeckt, versucht er eine Ausnahmeunterbrechung durchzuführen, wobei die entsprechenden Bits im Statuswort gesetzt werden. Ist der Ausnahmezustand dagegen durch das Kontrollwort maskiert, so führt der 8087 seine eigene Fehlerbehandlung durch. Er entscheidet dann, welche Maßnahmen zur Behebung des Fehlers notwendig sind, und führt die Verarbeitung mit einem zwangsweise substituierten Ergebnis im Register weiter. Dabei entsteht ein NAN-Ergebnis (Not A Number), wenn eine Operation nicht definiert ist, so wie z. B. die Quadratwurzel aus einer negativen Zahl. Unendlichkeit tritt auf, wenn das Ergebnis einer Operation zu groß ist, um in der Gleitpunktdarstellung repräsentiert zu werden.

Am anderen Ende der Gleitpunktdarstellung entstehen denormalisierte und unnormalisierte Zahlen. Wird eine Zahl nämlich so klein, daß sie im Exponentenfeld nicht länger dargestellt werden kann, so wird diese Zahl denormalisiert. Anstelle den Wert auf Null zu setzen, setzt der 8087 den Exponenten auf den kleinstmöglichen Wert und unnormalisiert den gebrochenen Teil der Zahl. Dies bedeutet außerdem, daß Genauigkeit verloren wird, da sich nun führende Nullen im gebrochenen Teil der Zahl befinden. Dennoch ist das solchermaßen denormalisierte Ergebnis immer noch genauer als die Alternative, nämlich Null. Dies ist ein Beispiel für einen Fall, in dem der 8087 weiterarbeitet und selbständig die bestmöglichen Ergebnisse aus einer schlechten Grundbedingung erzielt. Da die entsprechenden Flags zur Anzeige einer denormalisierten Zahl gesetzt sind, werden wir auf diesen Zustand hingewiesen. Auch das Bit im Ausnahmezustandsregister ist gesetzt, und es bleibt gesetzt, bis wir durch den Befehl FCLEX dieses Bit wieder löschen. Anhand dieser beiden Flags können wir in unserem Programm nun das Auftreten einer Fehlerbedingung erkennen und die jeweiligen Ergebnisse dann entsprechend genau untersuchen.

Funktionen und Transzendentes

Die letzte Gruppe der Befehle des 8087 führt mächtige mathematische Operationen durch. Diese Befehle erlauben es dem 8087, komplexere arithmetische Funktionen auszuführen, die beispielsweise logarithmische, exponentiale oder trigonometrische erfordern. In Abbildung 7.22 sehen wir eine Liste dieser Befehle. Wenn wir uns die Befehlsliste ansehen, werden wir entdecken, daß sich darauf nicht die gleichen Befehle befinden wie beispielsweise auf unserem Taschenrechner. Die

```
The IBM Personal Computer MACRO Assembler 01-01-83            PAGE    1-1
Figure 7.22  Stack Top Arithmetic Instructions

1                                              PAGE    ,132
2                                              TITLE   Figure 7.22  Stack Top Arithmetic Instructions
3                                      ENDIF
4        0000                          CODE    SEGMENT
5                                              ASSUME  CS:CODE,DS:CODE
6
7                                              FSQRT
8        0000  9B D9 FA            +           DB      09BH,0D9H,0FAH
9                                              FSCALE
10       0003  9B D9 FD            +           DB      09BH,0D9H,0FDH
11                                             FPREM
12       0006  9B D9 F8            +           DB      09BH,0D9H,0F8H
13                                             FRNDINT
14       0009  9B D9 FC            +           DB      09BH,0D9H,0FCH
15                                             FXTRACT
16       000C  9B D9 F4            +           DB      09BH,0D9H,0F4H
17                                             FABS
18       000F  9B D9 E1            +           DB      09BH,0D9H,0E1H
19                                             FCHS
20       0012  9B D9 E0            +           DB      09BH,0D9H,0E0H
21                                             FPTAN
22       0015  9B D9 F2            +           DB      09BH,0D9H,0F2H
23                                             FPATAN
24       0018  9B D9 F3            +           DB      09BH,0D9H,0F3H
25                                             F2XM1
26       001B  9B D9 F0            +           DB      09BH,0D9H,0F0H
27                                             FYL2X
28       001E  9B D9 F1            +           DB      09BH,0D9H,0F1H
29                                             FYL2XP1
30       0021  9B D9 F9            +           DB      09BH,0D9H,0F9H
31
32       0024                          CODE    ENDS
33                                             END
```

Abbildung 7.22 Arithmetische Befehle mit der Stackspitze

Designer des 8087 waren nämlich nicht in der Lage, alle gewünschten Funktionen innerhalb eines einzigen Chips unterzubringen, der bereits jetzt schon sehr komplex aufgebaut ist. Stattdessen wurde der Rechner mit einem Satz von Funktionen niedriger Ebene versehen, aus denen wir in einem Programm beispielsweise die Funktionen des Taschenrechners darstellen können. So gibt es keine trigonometrischen Funktionen wie Sinus oder Kosinus. Stattdessen existiert eine partielle Tangensfunktion,. Diese Funktion gibt einen Wert zurück, der dem Tangens eines Winkels entspricht. Aus diesem Wert kann ein Programm dann den Sinus, Kosinus, Tangens oder jeden anderen trigonometrischen Wert ableiten. Gehen wir den entgegengesetzten Weg, so gibt es eine partielle Arcustangensoperation, die einen bestimmten Wert benötigt und den Winkel für diesen Tangens zurückgibt. Diese partielle Funktion erlaubt es uns, den Arcussinus, den Arcuscosinus und andere Funktionen zu erzeugen, ohne daß diese als explizite Befehle vorliegen müßten.

Weiter unten können Sie eine Liste dieser Befehle finden, zusammen mit einer kurzen Erklärung der Arbeitsweise jedes einzelnen. Keiner dieser Befehle enthält dabei vom Programmierer definierte Operanden. Im allgemeinen arbeiten alle diese Befehle mit der Spitze des Stacks und möglicherweise auch mit dem Register ST1.

SQRT (Quadratwurzel)
ST ← Quadratwurzel(ST)
FSQRT ST darf nicht negativ sein
FSCALE (Scale)
ST ← ST * 2^{ST1}

Dieser Befehl ist für das Potenzieren notwendig. Die einzige andere verfügbare Exponentialfunktion hat nämlich nur einen beschränkten Bereich für den Wert des Exponenten. Der gezeigte Befehl multipliziert die Zahl mit ganzzahligen Zweierpotenzen. Später werden wir noch ein Beispiel für Zehnerpotenzen bringen.

FPREM (Partieller Divisionsrest)
ST ← ST partiell mod(ST1)

Der Befehl FPREM führt keine vollständige Modulodivision durch. Er reduziert jedoch den Inhalt des obersten Stackregisters um einen Wert von maximal 2^{64} in einem einzigen Befehl. Das Ergebnis des Befehls ist ein exakter Divisionsrest, wobei beispielsweise bei der Reduktion einer sehr großen Zahl mit einer sehr kleinen Basis eine verhältnismäßig große Zeitspanne verrinnen kann. Indem wir einen Maximalwert für die Reduktion bei jeder Befehlsausführung setzen, können wir Unterbrechungen während der Ausführung einer solchen totalen Modulodivision ermöglichen. Der Befehl FPREM setzt nämlich das Bedingungscodeflag C2 auf 1, wenn die Funktion nicht vollständig ausgeführt ist. Dagegen werden bei vollständiger Ausführung des Befehls die Statusflags C3, C1 und C0 mit den niedrigwertigen 3 Bits des Quotienten besetzt. Verwenden wir FPREM im Zusammenhang mit einer trigonometrischen Funktion, um beispielsweise den Radius eines Winkels einzuschränken, so ist es notwendig, den Oktanden des originalen Winkels festzulegen. Um diesen Befehl genauer zu erläutern, werden wir später auch noch ein trigonometrisches Beispiel anfügen.

FRNDINT (Runden auf ganze Zahlen)
ST ← Integer(ST)

Dieser Befehl rundet den Inhalt des obersten Stackregisters auf eine ganze Zahl. Die im Kontrollwort festgelegte aktuelle Rundungssteuerung bestimmt dabei die Richtung der Rundung.

FXTRACT **(Extract)**
ST ← gebrochener Teil von ST
ST1 ← Exponent von ST

Dieser Befehl bricht den Inhalt der aktuellen Stackspitze in seine Komponenten auf. Die aktuelle Stackspitze ist dabei das Funktionsargument. Nach Ausführung des Befehls ersetzt der Exponent des Arguments den Wert an der Spitze des Stacks. Außerdem wird der gebrochene Teil des Arguments in den Stack geschoben und dadurch die neue Stackspitze. Der Befehl ist außerdem komplementär zum Befehl FSCALE. Enthält die Stackspitze dabei einen bestimmten Wert und führen wir die Befehle FXTRACT und FSCALE nacheinander aus, so erhalten wir wieder den originalen Wert. Allerdings entfernt der Befehl FSCALE den Wert des Exponenten nicht aus dem Stack, so daß wir einen weiteren zusätzlichen Eintrag im Stack erhalten.

FABS **(Absolutwert)**
ST ← Absolutwert von ST

Dieser Befehl setzt das Vorzeichen der Stackspitze auf 0 (was eine positive Zahl bedeutet) und erstellt somit einen absoluten Wert.

FCHS **(Vorzeichenwechsel)**
ST ← −ST

Dieser Befehl ändert das Vorzeichen der Stackspitze.

Die folgenden Funktionen behandeln transzendente Operationen für Trigonometrie, Logarithmus und Potenzieren.

FPTAN **(Partieller Tangens)**
ST ← X
ST1 ← Y, wobei Y/X = TAN(Winkel)

Diese Funktion ist das Tor zu allen anderen trigonometrischen Funktionen wie Sinus, Kosinus und Tangens. Der Eingabewert hierzu befindet sich an der Spitze des Stacks und ist ein Winkel im Bogenmaß, der sich im Wertbereich 0<Winkel<PI/4 befinden muß. Mit dem Befehl FPREM können wir zusätzlich diesen Winkel auf das korrekte Maß reduzieren. Das Ergebnis ist eine Maßzahl, Y/X, die dem Tangens des Winkels entspricht. Y ersetzt außerdem den Winkelwert an der Spitze des Stacks und zusätzlich wird noch der Wert X in den Stack geschoben. Aus diesen Werten können dann alle anderen trigonometrischen Funktionen berechnet werden. So ist beispielsweise der Kosinus COS(Winkel)=X/SQR(X**2+Y**2)

FPATAN **(Partieller Arcustangens)**
ST ← Arctan(Y/X) = Arctan(ST1/ST)

Diese Funktion ist komplementär zur vorausgehenden Operation FPTAN. FPATAN berechnet nämlich den Winkel aus dem Verhältnis der beiden Werte in ST1 und

ST0. Dabei wird der X-Wert aus dem Stack gelesen und das Ergebnis, nämlich der Winkel, überschreibt den Wert Y als neue Stackspitze. Die Eingabewerte müssen dabei die Ungleichung

$0 < Y < X <$ Unendlich

erfüllen.

F2XM1 $2^X - 1$
$ST \leftarrow (2^{ST}) - 1$

Diese Funktion führt das Potenzieren durch. Sie erhebt dabei 2 in die in der Stackspitze angegebene Potenz. Der Eingabewert muß in den Grenzen $0 \leqq ST \leqq 0.5$ bleiben. Um mit einem Exponentwert größer 0.5 zu arbeiten, muß dieser Befehl in Kombination mit dem Befehl FSCALE verwendet werden. Mit der konstanten Funktion FLD können wir auch andere Werte als 2 potenzieren, indem wir folgende Formeln verwenden:

$$10^X = 2^{X \cdot Log2 10}$$
$$e^X = 2^{X \cdot Log2 e}$$
$$Y^X = 2^{X \cdot Log2 Y}$$

In einem weiteren Beispiel werden wir nun 10 in eine beliebige Potenz erheben.

FYL2X $(Y * Log_2 X)$
$ST \leftarrow Y \cdot Log_2 X = ST1 \cdot Log_2 ST$

Diese Funktion führt logarithmische Operationen durch. Dabei wird der Logarithmus zur Basis 2 aus der Stackspitze entnommen und dann mit dem Register ST1 multipliziert. Der Befehl FYL2X liest dann den Wert X aus dem Stack und ersetzt Y durch das Ergebnis der Operation. Die Parameter müssen dabei folgende Ungleichungen erfüllen:

$0 < X <$ Unendlich und $-$Unendlich $< 0 <$ Unendlich

Diese Funktion optimiert die Berechnung des Logarithmus zu einer anderen Basis als 2. Die Formel

$$Log_n 2 \times Log_2 X$$

ergibt den Logarithmus eines beliebigen Wertes zur Basis n. Wir können dann $Log_n 2$ als $1/(Log_2 n)$ berechnen.

FYL2XP1 $(Y * Log_2 X + 1)$
$ST \leftarrow Y \cdot Log_2(X + 1) = ST1 \cdot Log_2(ST + 1)$

Diese Funktion ist bis auf die Addition von 1 zu X identisch mit FYL2X. Die Funktion ist dabei stärker an X gebunden und für Berechnungen des Logarithmus gedacht, bei denen X sich sehr nahe an 1 befindet. Die Funktion erhöht dabei vor allem die Genauigkeit, vorausgesetzt

$$0 < ABS(X) < 1 - \frac{\sqrt{2}}{2}$$

Beispiele

Das folgende Kapitel soll Ihnen den Gebrauch des 8087 durch Beispiele, nicht durch Erklärungen, nahebringen. Wir bringen dabei Beispiele für die gängigsten Operationen mit dem 8087. Alle diese Beispiele sind einfach. Das heißt, wir versuchen dabei nicht, alle möglichen Fehlerbedingungen zu behandeln oder irgendwelche speziellen Zahlen herauszufinden, die der 8087 außerdem bearbeiten könnte. Der 8087 ist natürlich in der Lage, solche Dinge zu bearbeiten, doch sind dies bereits sehr fortgeschrittene Operationen. Unsere Beispiele sollen Ihnen dagegen zeigen, wie Sie mit dem 8087 umgehen, ohne daß Sie ihn vorher bereits gekannt haben. Wenn Sie diese einfachen Techniken einmal beherrschen, dann können Sie an diesen Beispielen die nötigen Verfeinerungen anbringen, die sie dann zu wirklich allgemein verwendbaren Routinen machen.

Potenzen von 10

Das Quellprogramm in Abbildung 7.23 ist unser erstes Beispiel. Dieses Programm druckt die kurze Gleitpunktdarstellung der Potenzen von 10 von 10^3 bis 10^{39} aus. Wie wir bereits im Abschnitt über die Datendarstellung gehört haben, verfügt der IBM-Makroassembler über keine Möglichkeit, Gleitpunktzahlen direkt einzugeben. Mit der von unserem Programm erzeugten Tabelle ist es Ihnen dann ein einfaches, die Potenzen von 10 als Konstanten einzugeben. Sie können nämlich aus der Tabelle die korrekten hexadezimalen Werte entnehmen und sie dann in Ihr Programm einfügen.

Unser Programm erzeugt nur jede dritte Potenz von 10 und wir verwenden dazu die kurze Gleitpunktdarstellung. Wollen wir dagegen mit größeren Zahlen arbeiten oder benötigen wir eine größere Präzision, dann sollten wir hier lange Gleitpunktzahlen verwenden und außerdem jede Potenz von 10 erzeugen. Wir wollen Ihnen das allerdings nur als zusätzliche Übung überlassen.

Der Hauptzweck des Beispiels ist nämlich eine Einführung in den 8087 und seine Arbeitsweise. Unser Programm ist dabei ein vollkommen selbständiges Programm, das als .EXE-Datei direkt ausgeführt werden kann. Bevor wir uns dem Programm selbst zuwenden, halten wir fest, daß unser Programm ein Stacksegment enthält, das für eine .EXE-Datei notwendig ist. Die Datenbereiche erscheinen dabei am Anfang des Codesegments und unser Programm beginnt mit dem Label CALCULATE_POWER. Sehen wir nun nach hinten zum END-Befehl, so können wir sehen, daß CALCULATE_POWER tatsächlich der erste Befehl ist, da er auch im END-Statement aufgeführt wird.

Der erste Teil des Programms bewirkt die Initialisierung. Dabei schiebt unser Programm die Rückkehradresse für eine .EXE-Datei in den Stack, bevor wir das DS-Register auf den Anfang des CODE-Segments setzen. Mit dem Befehl FINIT initialisieren wir außerdem den 8087, was bei diesem einem Hardware-Reset entspricht. Dabei werden im 8087 alle standardmäßig vorgesehenen Fehlerbehandlungsroutinen aktiviert, was für alle Beispiele in diesem Buch sehr praktisch ist. Der Befehl FINIT löscht außerdem den Registerstack des 8087 und stellt damit alle acht Stack-

positionen zu unserer Benutzung bereit. Wir sollten deshalb den Befehl FINIT nur beim tatsächlichen Start eines Programms verwenden. So sollte der Befehl z. B. keinesfalls ein Teil eines Unterprogramms für den 8087 sein.

Der nächste Befehl lädt den Wert 1000 in das Register ST1 und den Wert 1 in das Register ST0. Alle weiteren Operationen mit dem 8087 benützen diese beiden Stackregister. Dabei enthält ST0 die aktuelle Potenz von 10, während ST1 10^3 enthält. Wir werden diesen Wert in ST1 verwenden, um damit nach jeder Programmschleife den Wert in ST0 zu erhöhen. Die Integervariable POWER enthält immer die Potenz von 10, die sich aktuell in ST0 befindet.

Beim Label POWER_LOOP wird ST0 mit ST1 multipliziert (das 1000 enthält), um so ST0 um 10^3 zu erhöhen. Der Befehl FST legt das Ergebnis dann an einer Speicherstelle ab. Der restliche Teil von POWER_LOOP gibt das Ergebnis der Operation aus. Das Unterprogramm TRANSLATE konvertiert dabei einen 1-Byte Hexadezimalwert in einen zwei Zeichen langen ASCII-String, so daß er vom Programm ausgedruckt werden kann. Der aktuelle Inhalt von POWER, also die laufende Potenz von 10, und der vom 8087 gespeicherte hexadezimale String werden beide in ASCII konvertiert. Eine DOS-Funktion gibt diesen String dann auf dem Bildschirm aus. Die Schleife in POWER_LOOP wird solange durchlaufen, bis der letzte ausgegebene Wert größer als 10^{38} ist. Wir wählten diesen Wert 10^{38}, da er der größte darstellbare Wert für kurze Gleitpunktzahlen ist. Hätten wir dagegen für die Darstellung der Zahlen die lange Gleitpunktform gewählt, so wäre die maximale darstellbare Zahl 10^{308} gewesen. Der letzte Teil in Abbildung 7.23 zeigt schließlich noch den Output des Programms wie er auf dem Bildschirm erscheint.

```
        PAGE    ,132
        TITLE   Figure 7.23 Powers of Ten
IF1
INCLUDE 87MAC.LIB
ENDIF
STACK   SEGMENT STACK
        DW      64 DUP(?)
STACK   ENDS
CODE    SEGMENT
        ASSUME  CS:CODE
POWER_OF_TEN    DD      ?               ; Data area for 10**x, short real
OUTPUT_POWER    DB      2 DUP(' ')      ; Buffer for power value
                DB      'H    '
OUTPUT_STRING   DB      8 DUP(' ')      ; Hexadecimal string output
                DB      'H',13,10,'$'   ; String ending
POWER           DB      0               ; Current power of ten
THOUSAND        DW      1000            ; Constant
CONTROL_87      DW      03BFH
CALCULATE_POWER PROC    FAR
        PUSH    DS                      ;Set return address
        MOV     AX,0
        PUSH    AX
        PUSH    CS
        POP     DS
        ASSUME  DS:CODE                 ; Addressing to data area
        FINIT                           ; Initialize the 8087
        FILD    THOUSAND                ; Load 10**3
        FLD1                            ; Load starting value
POWER_LOOP:
        FMUL    ST1                     ; Multiply st0 * st1
        FST     POWER_OF_TEN            ; Store into ram
        ADD     POWER,3                 ; Update power of ten
        MOV     AL,POWER                ; Get power
        MOV     BX,OFFSET OUTPUT_POWER
        CALL    TRANSLATE
        MOV     CX,4
        MOV     BX,OFFSET OUTPUT_STRING
        MOV     SI,OFFSET POWER_OF_TEN+3
        STD                             ; Backwards
VALUE_OUTPUT:
        LODSB                           ; Get byte of power
        CALL    TRANSLATE               ; Write to output string
        LOOP    VALUE_OUTPUT            ; Do it for all bytes of string
```

```
        MOV     DX,OFFSET OUTPUT_POWER
        MOV     AH,9
        INT     21H
        CMP     POWER,38
        JB      POWER_LOOP
        FCOMPP                          ; Pop two items from stack
        RET
CALCULATE_POWER ENDP

TRANSLATE       PROC    NEAR
        PUSH    AX                      ; Save input value
        PUSH    CX
        MOV     CL,4                    ; Move high nybble to
        SHR     AL,CL                   ;  low nybble
        POP     CX
        CALL    XLAT_OUTPUT             ; Output low nybble
        POP     AX                      ; Parameter back
        CALL    XLAT_OUTPUT             ; Output high nybble
        RET
TRANSLATE       ENDP

ASCII_TABLE     DB      '0123456789ABCDEF'
XLAT_OUTPUT     PROC    NEAR
        AND     AL,0FH                  ; Isolate low nybble
        PUSH    BX
        MOV     BX,OFFSET ASCII_TABLE   ; Translate table
        XLAT    ASCII_TABLE             ; Convert to ASCII
        POP     BX
        MOV     [BX],AL                 ; Store in output string
        INC     BX                      ; Point to next in string
        RET
XLAT_OUTPUT     ENDP
CODE    ENDS
        END     CALCULATE_POWER

        Figure 7.23(a)
```

```
A>PRINT10
03H     447A0000H
06H     49742400H
09H     4E6E6B28H
0CH     5368D4A5H
0FH     58635FA9H
12H     5D5E0B6BH
15H     6258D727H
18H     6753C21CH
1BH     6C4ECB8FH
1EH     7149F2CAH
21H     76453719H
24H     7B4097CEH
27H     7F800000H

        Figure 7.23(b)
```

Abbildung 7.23 (a) Potenzen von 10; (b) Ausgabe des Programms

Wir sollten uns dabei den TRANSLATE-Teil unseres Programms besonders ansehen, selbst wenn er keine einzige Anweisung für den 8087 enthält. Dieser Programmteil ist nämlich ein Beispiel dafür, wie wir Zahlenwerte zur Ausgabe vorbereiten. Im besonderen konvertiert dabei der Befehl XLAT das hexadezimale Halbbyte (0 bis 0FH) in den korrekten ASCII-Zeichenwert („0" bis „F") zur Druckausgabe. Wir können nämlich nicht einfach einen Wert auf dieses Halbbyte addieren, da die Zeichen „A" bis „F" den Zeichen „0" bis „9" im ASCII-Zeichensatz nicht direkt folgen. Der XLAT-Befehl erledigt diese Sache für uns. Wir werden später eine ähnliche Methode verwenden, wenn wir Gleitpunktzahlen zur Ausgabe in Dezimalzahlen umwandeln müssen.

Zehn in der Xten Potenz

Das zweite Beispiel für die Anwendung des 8087 führt uns wesentlich tiefer in seine Arbeitsweise ein. Dieses Programm ist ein Unterprogramm. Und in der Tat werden wir dieses Unterprogramm auch noch in späteren Programmen verwenden. Das Programm nimmt dabei an, daß der Eingabeparameter einen Wert X an der Spitze des Stacks enthält. Nach Rückkehr aus dem Unterprogramm ist der Wert an der Stackspitze 10. Das Quellprogramm sehen Sie in Abbildung 7.24.

```
        PAGE    ,132
        TITLE   Figure 7.24 Calculate 10**ST
IF1
INCLUDE 87MAC.LIB
ENDIF
CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,DS:CODE
        PUBLIC  TEN_TO_X
OLD_CW  DW      ?
NEW_CW  DW      ?
;-----------------------------------------------;
; This routine takes the top element of         ;
;  the 8087 stack, and raises ten to that power ;
; Input -- ST0 is X                             ;
; Output -- ST0 is 10**X                        ;
; This routine uses two stack positions plus    ;
;   the parameter, a total of three.            ;
;-----------------------------------------------
TEN_TO_X        PROC    NEAR
                                ;-----ST0-------;-----ST1-------;---ST2-------
                                ;      X        ;      ?        ;   ?
        FLDL2T                  ; LOG2(10)      ;      X        ;   ?
        FMULP   ST1,ST0         ;X LOG2(10) = E ;      ?        ;   ?
        FNSTCW  OLD_CW          ;---------------;---------------;-------------
        FWAIT                   ; Get the current status word
        MOV     AX,OLD_CW       ; Save it
        AND     AX, NOT 0C00H   ; Set rounding control to
        OR      AX, 0400H       ;  round towards -infinity
        MOV     NEW_CW,AX
        FLDCW   NEW_CW          ;---------------;---------------;-------------
        FLD1                    ;       1       ;       E       ;       ?
        FCHS                    ;      -1       ;       E       ;       ?
        FLD     ST1             ;       E       ;      -1       ;       E
        FRNDINT                 ; INT(E) = I    ;      -1       ;       E
        FLDCW   OLD_CW          ;               ;               ;
        FXCH    ST2             ;       E       ;      -1       ;       I
        FSUB    ST0,ST2         ; E - I = F     ;      -1       ;       I
        FSCALE                  ; F*2**-1 = F/2 ;      -1       ;       I
        F2XM1                   ;(2**F/2)-1     ;      -1       ;       I
        FSUBRP  ST1,ST0         ; 2**F/2        ;       I       ;       ?
        FMUL    ST0             ; 2**F          ;       I       ;       ?
        FSCALE                  ;(2**F)*(2**I)  ;       I       ;       ?
        FXCH    ST1             ;       I       ;   2**(I+F)    ;       ?
        FCOMP                   ;   2**(I+F)    ;       ?       ;       ?
        RET                     ;  10**X        ;       ?       ;       ?
TEN_TO_X        ENDP
CODE    ENDS
        END
```

Abbildung 7.24 Berechnung von 10**ST

Es gibt keine 8087-Anweisung, mit der wir 10 in eine beliebige Potenz erheben könnten. Doch wir können dies mit der Zahl 2. Dazu benutzen wir die folgende Formel:

10X = 2**(X*Log2(10))**

Die ersten beiden 8087-Befehle formen dabei den neuen Exponenten für 2. Das Programm lädt sodann die Konstante Log2(10) und multipliziert damit den Eingabewert X, wodurch wir den gesuchten Exponenten von 2, den wir als E bezeichnen, erhalten. Halten wir fest, daß wir in unserem Beispiel die Kommentare dazu verwenden, die benutzten Stackpositionen des 8087 anzugeben. Das Zeichen „?" gibt dabei an, daß der Wert des betreffenden Stackelements unbekannt ist. Insgesamt werden von unserem Unterprogramm drei Stackplätze belegt, weshalb wir auch drei Kommentarspalten verwendet haben.

Nachdem der benötigte Exponent von 2 ermittelt ist, müssen wir feststellen, daß es keine 8087-Anweisung gibt, die den Rest der Aufgabe in einem Schritt bewältigen könnte. Der Befehl F2XM1 erhebt 2 zwar in die Potenz X, aber nur für Werte von $X \leqq 1/2$. Dies bedeutet, daß wir den Exponenten E in einen ganzzahligen und einen gebrochenen Teil zerlegen müssen. Mit FSCALE können wir 2 in eine beliebige ganzzahlige Potenz erheben, und mit F2XM1 können wir den Rest der Aufgabe bewältigen.

Bevor wir nun E aufspalten, müssen wir noch etwas Ordnung schaffen. Die nächste Befehlsfolge für den 8087 liest das Kontrollwort und setzt die Rundungskontrolle

auf Abrunden. Dies stellt sicher, daß beim Ermitteln des ganzzahligen Teils des Exponenten nötigenfalls in Richtung negative Unendlichkeit gerundet wird, wodurch der gebrochene Teil des Exponenten immer die von F2XM1 benötigte positive Zahl ist.

Beachten Sie dabei den Befehl FWAIT nach FNSTCW. Wir müßten nämlich zum Speichern des Steuerworts eigentlich nicht auf das Ende der Multiplikation warten, da diese den darin enthaltenen Wert nicht verändert. Allerdings müssen wir vor dem Lesen und etwaigen Verändern des Kontrollworts sicherstellen, daß es vom 8087 auch übermittelt wurde. Zu diesem Zweck wurde hier der Befehl FWAIT eingefügt.

Nachdem die Rundungskontrolle gesetzt ist, rundet der Befehl FRNDINT den Exponenten E auf eine ganze Zahl ab. Da wir aber auch den ursprünglichen Wert von E im Stack sichergestellt haben, können wir den ganzzahligen Teil von E subtrahieren und so den gebrochenen Teil des Exponenten ermitteln. So verfügen wir nun über E=I+F, und können wie folgt weiterfahren:

2**E = 2**I*2**F

Allerdings benötigen wir dazu noch etwas. Der gebrochene Teil F könnte größer als 1/2, und deshalb als Argument für F2XM1 ungeeignet sein. Um F nun durch 2 zu teilen, also F/2 zu ermitteln, verwenden wir den Wert −1, den wir bereits früher in den Stack gespeichert haben. Durch Benutzung des Befehls FSCALE, der ST0 mit 2 in der Potenz von ST1 multipliziert, erhalten wir das gewünschte Ergebnis. Da ST1 −1 enthält, ist die Folge der Operation eine Multiplikation mit 1/2. Jetzt können wir sicher sein, daß ST0 kleiner als 1/2 ist.

Der Befehl F2XM1 erhebt nun 2 in die Potenz F/2, und die −1 im Stack wird durch das von F2XM1 miterzeugte Ergebnis −1 überschrieben. Durch eine reverse Subtraktion (mit POP) entfernen wir den Wert −1 aus dem Stack. Darauf wird 2**(F/2) mit sich selbst multipliziert, was zu 2**F in ST0 führt. Da sich der ganzzahlige Teil des Exponenten nun in ST1 befindet, nimmt FSCALE 2**I und multipliziert diesen Wert mit dem in ST0 enthaltenen 2**F, wodurch wir das gewünschte Ergebnis erhalten. FCOMP entfernt nun noch vor der Rückkehr aus dem Unterprogramm den Wert I aus dem Stack.

Gleitpunktausgabe

Mit dem nächsten Unterprogramm, das wir besprechen wollen, werden wir die Konvertierung der Daten an der Stackspitze in einen ausgebbaren Zeichenstring durchführen. Wir wollen das Programm dabei so schreiben, daß es die Daten aus der Stackspitze entnimmt und auf den Bildschirm ausgibt. Wir werden diese Routine in den nächsten beiden Beispielen verwenden, um die Ergebnisse der darin verwendeten Programm auszugeben. Die Liste des Unterprogramms finden Sie in Abbildung 7.25.

Das Unterprogramm führt eine recht simple Bestimmung des korrekten ASCII-Strings durch. Ist der Eingabewert nämlich NAN oder unendlich oder sonst eine besondere Zahl des 8087, so wird ein falsches Ergebnis erzeugt. Der erste Teil unserer Routine wäre dabei ein gutes Beispiel für die Verwendung des FXAM-Befehls, der den Typ des Operanden der Stackspitze ermittelt. Wir übergehen dies

aber und nehmen an, daß die Eingabewerte für unsere Routine immer ordnungsgemäß aufgebaut sind.

Auch wird die Formatierung der Ausgabe nur sehr sparsam durchgeführt. Dabei haben wir immer ein Vorzeichen (Leerstelle oder Minuszeichen) und einen einziffrigen ganzzahligen Teil. Nach dem Dezimalpunkt folgen noch acht Ziffernstellen. Auf das Zeichen „E" folgen das Vorzeichen des Exponenten und eine dreiziffrige Angabe der jeweiligen Potenz von 10. Die Ausgabe unseres Unterprogramms ist längst nicht so schön, wie wir es uns wünschen könnten, doch gestattet sie es uns, die Ergebnisse unserer Programme vernünftig lesen zu können. Um die Ausgabe schöner zu gestalten, müßten wir eine Menge zusätzlicher Befehle einfügen, und nur wenige von diesen würden unser Verständnis des 8087 erhöhen.

Das Umsetzprogramm arbeitet etwa folgendermaßen: Zuerst wird die Größe der Zahl festgestellt. Die Zahl 1234 hat z.B. die Größe 3, d.h. sie bewegt sich zwischen 10^3 und 10^4. Haben wir nun diese Größe korrekt bestimmt, sichert unser Programm diesen Wert (er entspricht dem Exponenten des Ergebnisses) und dividiert das ursprüngliche Ergebnis durch 10 in der Potenz dieser Zahl. Dies wandelt den Eingabewert in eine Zahl zwischen 1 und 10 um. Nun wird die Zahl mit 10^8 multipliziert. Das Ergebnis in BCD-Form umfaßt neun Ziffernstellen. Die höchste Stelle ist dabei der ganzzahlige Teil, die restlichen acht Stellen sind die Stellen hinter dem Dezimalpunkt.

Im ersten Teil des Programms wird die richtige Größe des Eingabeparameters ermittelt. Dabei erhalten wir mit folgender Formel den Logarithmus zur Basis 10 einer beliebigen Zahl:

$$\mathbf{Log10(X) = Log2(X)/Log2(10)}$$

Durch Setzen der Rundungskontrolle runden wir diese Zahl in Richtung — unendlich ab. Im vorhergehenden Beispiel, in dem wir 10^X berechneten, erhielten wir bereits den korrekten Multiplikationsfaktor für den Inputwert. Dies reduziert den Bereich der Eingabezahl auf 1 bis 10. Wir verwenden den konstanten Wert TENB (der die Zahlenkonstante 10^8 darstellt), um unsere Zahl in den richtigen Größenbereich zu bringen. Und schließlich wandeln wir noch mittels des Befehls FBSTP die beiden Zahlen in BCD-Darstellung um. Beim erstenmal verwenden wir den Befehl dabei dazu, die neun Ziffern des gebrochenen Teils der Zahl festzulegen, und beim zweitenmal, um die drei Ziffern des Exponenten darzustellen.

```
        PAGE    ,132
        TITLE   Figure 7.25 Floating point to ASCII Conversion
IF1
INCLUDE 87MAC.LIB
ENDIF

CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,DS:CODE,ES:CODE
        EXTRN   TEN_TO_X:NEAR
OLD_CW          DW      ?
NEW_CW          DW      ?
EXPONENT        DW      ?
BCD_RESULT      DT      ?
BCD_EXPONENT    DT      ?
TEN8            DD      100000000
PRINT_STRING    DB      '  .        E    ',10,13,'$'
        PUBLIC  FLOAT_ASCII
;-----------------------------------------------;
; This routine takes the top element of the     ;
;  8087 stack and displays the floating point   ;
;  value.                                       ;
; Input -- ST0 of the 8087                      ;
; Output -- Value is displayed, stack is popped ;
;-----------------------------------------------;
```

```
FLOAT_ASCII     PROC    NEAR
                                ;------ST0------;------ST1------;------ST2----
                                ;       X       ;               ;
        FLD     ST0             ;       X       ;       X       ;       ?
        FABS                    ;      |X|      ;       X       ;
        FLD1                    ;       1       ;       X       ;       X
        FXCH    ST1             ;       X       ;       1       ;       X
        FYL2X                   ;    LOG2(X)    ;       X       ;       ?
        FLDL2T                  ;    LOG2(10)   ;    LOG2(X)    ;       X
        FDIVRP  ST1,ST0         ;E=LOGX/LOG10   ;       X       ;       ?
        FNSTCW  OLD_CW          ;               ;               ;
        FWAIT                   ;               ;               ;
        MOV     AX,OLD_CW       ;               ;               ;
        AND     AX, NOT 0C00H   ;               ;               ;
        OR      AX, 0400H       ;               ;               ;
        MOV     NEW_CW,AX       ;               ;               ;
        FLDCW   NEW_CW          ;               ;               ;
        FRNDINT                 ;  I= INT(E)    ;       X       ;       ?
        FLDCW   OLD_CW          ;               ;               ;
        FIST    EXPONENT        ;       I       ;       X       ;       ?
        FCHS                    ;      -I       ;       X       ;       ?
        CALL    TEN_TO_X        ;  10 **(-I)    ;       X       ;       ?
        FMULP   ST1,ST0         ;  X/10**I      ;       ?       ;       ?
        FIMUL   TEN8            ; ADJUSTED FRAC ;       ?       ;       ?
        FBSTP   BCD_RESULT      ;       ?       ;       ?       ;       ?
        FILD    EXPONENT        ;       I       ;       ?       ;       ?
        FBSTP   BCD_EXPONENT    ;       ?       ;       ?       :       ?
;----- Display the values stored as BCD strings

        CLD
        MOV     DI,OFFSET PRINT_STRING          ; Point at output string
        MOV     AL,BYTE PTR BCD_RESULT+9
        CALL    PRINT_SIGN                      ; Print the sign
        MOV     AL,BYTE PTR BCD_RESULT+4
        CALL    PRINT_NYBBLE                    ; Print the leading digit
        MOV     AL,'.'                          ; The decimal point
        STOSB
        MOV     BX,OFFSET BCD_RESULT+3
        MOV     CX,4                            ; Loop through the 8
DO_BYTE:                                        ;  digits following
        CALL    PRINT_BYTE                      ;  the decimal point
        LOOP    DO_BYTE
        MOV     AL,'E'                          ; Exponent indicator
        STOSB
        MOV     AL,BYTE PTR BCD_EXPONENT+9
        CALL    PRINT_SIGN                      ; Print exponent sign
        MOV     AL,BYTE PTR BCD_EXPONENT+1
        CALL    PRINT_NYBBLE                    ; Print first digit
        MOV     BX,OFFSET BCD_EXPONENT
        CALL    PRINT_BYTE                      ; Last two digits of exp
        MOV     DX,OFFSET PRINT_STRING
        MOV     AH,9
        INT     21H                             ; Use DOS to print the string
        RET
FLOAT_ASCII     ENDP
;----- This routine prints a ' ' or '-'

PRINT_SIGN      PROC    NEAR
        CMP     AL,0                    ; Test for minus sign
        MOV     AL,' '                  ; Positive
        JZ      POSITIVE
        MOV     AL,'-'                  ; Mark as negative
POSITIVE:
        STOSB                           ; Put sign in the print string
        RET
PRINT_SIGN      ENDP

;------ This routine prints the two decimal
;------  digits pointed to by [BX]

PRINT_BYTE      PROC    NEAR
        MOV     AL,[BX]                 ; Get BCD byte
        PUSH    CX
        MOV     CL,4
        SHR     AL,CL                   ; Shift high nybble to low portion
        POP     CX
        CALL    PRINT_NYBBLE            ; Print high nybble
        MOV     AL,[BX]                 ; Get original value back
        CALL    PRINT_NYBBLE            ; Print low nybble
        DEC     BX                      ; Move to next byte in string
        RET
PRINT_BYTE      ENDP

;------ Print as decimal the AL value

PRINT_NYBBLE    PROC    NEAR
        AND     AL,0FH                  ; Isolate low nybble
        ADD     AL,'0'                  ; Convert to ASCII value
        STOSB                           ; Store in print string
        RET
PRINT_NYBBLE    ENDP
CODE    ENDS
        END
```

Abbildung 7.25 Umwandlung von Gleitpunkt auf ASCII

Im restlichen Teil des Unterprogramms wird noch die nötige Zeichenmanipulation durchgeführt, um die BCD-Darstellung in einen Zeichenstring umzuformen. Dazu werden Vorzeichen und Wert des Exponenten festgestellt und ausgegeben. Die BCD-Bytes werden also entpackt und in ASCII-Zeichen konvertiert. Das Entpacken geschieht dabei durch die Routine PRINT_BYTE, während die Routine PRINT_ NIBBLE die Umsetzung in ASCII-Zeichen durchführt. Halten wir fest, daß wir dieses Mal den Befehl XLAT nicht benötigen, da sich die Zahlen in dem Bereich zwischen 0 und 9 befinden (ist allerdings die Eingabezahl eine der undefinierten Zahlen, so wird auch der Ausgabestring einige seltsame Zeichen enthalten).

Unser Programm gibt alle Zahlen korrekt aus, die sich im Bereich der darstellbaren langen Gleitpunktzahlen befinden. Jede Zahl, die über diese Darstellung hinausgeht (beispielsweise 10^{1234}), wird in ihrem Exponentenfeld auf drei Ziffern reduziert. Natürlich könnten wir unser Programm so verändern, daß es vier Ziffern im Exponentenfeld bearbeiten kann. Es gibt allerdings eine Zahl, die unser Programm korrekt behandelt, deren Ausgabebild Sie aber sicherlich verändern wollen. Ist nämlich unsere Zahl 0, so wird das Ergebnis als 0.00000000E-932 ausgegeben. Dies geschieht durch Normalisierung des Exponenten im 8087. Der 8087 stellt nämlich die Zahl 0 mit dem kleinstmöglichen Exponenten (—4932) und mit einer Mantisse von 0 dar. Setzt unser Programm nun diese Zahl in ASCII-Zeichen um, so werden Mantisse und Exponent korrekt ausgegeben (abgesehen von der Tatsache, daß der Exponent auf drei Stellen verkürzt ist). Sicherlich wollen Sie nun in diesem Sonderfall den Exponenten besonders behandeln. So könnten wir in unser Programm einen Test auf 0 einfügen (beispielsweise mit dem Befehl FTST), und zwar ganz am Anfang des Programms, und so diesen Sonderfall behandeln. Auch diese Übung soll Ihrer eigenen Initiative überlassen bleiben.

Quadratische Gleichung

Im weiteren behandeln wir nun zwei Beispiele, die unsere bereits erstellte Routine zur Ausgabe von Gleitpunktzahlen verwenden werden. Das erste Beispiel ist dabei die Lösung einer quadratischen Gleichung. Gegeben sei die Formel:

0 = A*X**2 + B*X + C

Diese Gleichung wollen wir nun lösen. Aus dem Gymnasium kennen wir bereits

X = (—B + — SQR(B**2—4*A*C))/2*A

Das Programm zum Lösen dieser Gleichung ist ganz einfach und in Abbildung 7.26 gezeigt. Wir nehmen dabei an, daß die drei Parameter A, B und C als Integerzahl im Programm gespeichert sind. Sollten Sie unser Programm für mehr als Beispielzwecke verwenden wollen, dann müssen Sie selbstverständlich eine andere Art der Dateneingabe vorsehen.

In unserem Beispiel verwenden wir keine komplexe Arithmetik. Als erstes erfolgt eine Überprüfung auf eine negative Diskriminante (B**2-4*A*C). Ist dieser Wert negativ, so beendet unser Programm mit einer Fehlermeldung. Es gibt allerdings keinen Grund, warum Sie nicht auch komplexe Arithmetik in Ihrem Programm verwenden könnten.

```
        PAGE    ,132
        TITLE   Figure 7.26 Quadratic Equation roots
IF1
INCLUDE 87MAC.LIB
ENDIF
STACK   SEGMENT STACK
        DW      64 DUP(?)
STACK   ENDS

CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,DS:CODE,ES:CODE
        EXTRN   FLOAT_ASCII:NEAR
A       DW      1
B       DW      -5
C       DW      6
STATUS  DW      ?
FOUR    DW      4
TWO     DW      2
ERROR_MSG       DB      'Roots are imaginary',10,13,'$'

QUADRATIC       PROC    FAR
        PUSH    DS              ; Return address for .EXE
        SUB     AX,AX
        PUSH    AX
        MOV     AX,CS
        MOV     DS,AX
        MOV     ES,AX

        FINIT                   ;------ST0------;-----ST1-------
        FILD    B               ;       B       ;
        FMUL    ST0             ;      B**2     ;
        FILD    A               ;       A       ;     B**2
        FIMUL   FOUR            ;      4*A      ;     B**2
        FIMUL   C               ;     4*A*C     ;     B**2
        FSUBRP  ST1,ST0         ;   D=B**2-4AC  ;      ?
        FTST                    ;
        FSTSW   STATUS
        FWAIT
        MOV     AH,BYTE PTR STATUS+1
        SAHF
        JB      IMAGINARY
        FSQRT                   ; SQR(D)        ;
        FLD     ST0             ; SQR(D)        ;   SQR(D)
        FCHS                    ; -SQR(D)       ;   SQR(D)
        FIADD   B               ; B-SQR(D)      ;   SQR(D)
        FCHS                    ; -B+SQR(D)     ;   SQR(D)
        FXCH    ST1             ;   SQR(D)      ; -B+SQR(D)
        FIADD   B               ;  B+SQR(D)     ; -B+SQR(D)
        FCHS                    ;N1= -B-SQR(D)  ;N2= -B+SQR(D)
        FIDIV   A               ;  N1/A         ;      N2
        FIDIV   TWO             ; ROOT1 =N1/2A  ;      N2
        CALL    FLOAT_ASCII     ;       N2
        FIDIV   A               ;    N2/A       ;
        FIDIV   TWO             ; ROOT2 = N2/2A ;
        CALL    FLOAT_ASCII     ;        ?      ;
        RET
IMAGINARY:
        MOV     DX,OFFSET ERROR_MSG
        MOV     AH,9
        INT     21H             ; Display error message
        RET
QUADRATIC       ENDP
CODE    ENDS
        END     QUADRATIC
```

Abbildung 7.26 Quadratische Gleichung

Wir haben uns allerdings entschlossen, dies im Beispiel nicht zu tun. Sie sollten sich allerdings bewußt sein, daß der 8087, falls Sie es wünschen, nicht automatisch komplexe oder imaginäre Arithmetik durchführt. Sie müssen also in diesem Fall ein Programm schreiben, das den realen und den imaginären Teil von komplexen Zahlen getrennt bearbeitet.

Der Befehl FTST bewirkt einen Test auf eine negative Diskriminante. Der Befehl wirkt genauso wie ein Vergleichsbefehl, in dem ein fester Wert von 0 eingebaut ist. Halten wir fest, daß unser Programm das Statuswort im Speicher ablegt und es dann in das Flagregister des 8088 überträgt. Dies ermöglicht einen schnellen Test (JB für Jump Below), um festzustellen, ob die Diskriminante kleiner als Null ist. Der Rest des Programms dient zum Berechnen der beiden Wurzeln der Gleichung. Unser Programm nützt dabei die Tatsache aus, daß sich die beiden Parameter im Speicher befinden.

Diese Technik minimiert den Stackbedarf des 8087. Sollten Sie allerdings das Programm modifizieren wollen, z.B. um es als Unterprogramm für einige andere Programme einzusetzen, dann könnte es wünschenswert sein, die Parameter auf dem Stack des 8087 zu übergeben. Dieser Weg würde allerdings bedeuten, daß Sie die Art, in der das Programm einige Werte behandelt, an einigen Stellen verändern müßten.

Sinus eines Winkels

Das letzte Beispiel, bei dem wir den 8087 verwenden, ist die Berechnung des Sinus eines Winkels. Der 8087 verfügt dabei nicht über einen eigenen Befehl, um die Sinusfunktion auszuführen. Das Beste, was der 8087 in diesem Fall offerieren kann, ist der Befehl FPTAN, die Anweisung für einen partiellen Tangens. Wir werden diesen Befehl verwenden, zusammen mit dem Befehl FPREM (partieller Rest), um die Sinusoperation durchzuführen.

Das Programm zur Berechnung des Sinus zeigen wir in Abbildung 7.27. Dabei wird der Sinus eines jeden Winkels zwischen 1/2 bis 6 für jeden halben Bogengrad berechnet und ausgegeben. Wir erzeugen dabei einen Output, der dem eines BASIC-Programms sehr stark ähnelt.

10 FOR X = .5 TO 6.0 STEP 0.5

20 PRINT SIN(X)

30 NEXT X

Für die Ausgabe unserer Zahl verwenden wir die Gleitpunktausgaberoutine, die wir in Abbildung 7.25 bereits gezeigt haben.

```
        PAGE    ,132
        TITLE   Figure 7.27 Sine computation
IF1
INCLUDE 87MAC.LIB
ENDIF
STACK   SEGMENT STACK
        DW      64 DUP(?)
STACK   ENDS
CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,DS:CODE,ES:CODE
        EXTRN   FLOAT_ASCII:NEAR
NUM_ANGLE       DW      1
DEN_ANGLE       DW      2
STATUS  DW      ?
FOUR    DW      4
C3      EQU     40H
C2      EQU     04H
C1      EQU     02H
C0      EQU     01H
ERROR_MSG       DB      'Angle is too large',10,13,'$'
SIN     PROC    FAR
PUSH    DS
SUB     AX,AX
PUSH    AX
MOV     AX,CS
MOV     DS,AX
MOV     ES,AX
DO_AGAIN:
FINIT                           ;------ST0------;-----ST1-------
FILD    NUM_ANGLE               ;               ;
FIDIV   DEN_ANGLE               ;  X= ANGLE     ;
FLDPI                           ;      PI       ;       X
FIDIV   FOUR                    ;    PI/4       ;       X
FXCH                            ;      X        ;     PI/4
FPREM                           ;      R        ;     PI/4
FSTSW   STATUS
FWAIT
MOV     AH,BYTE PTR STATUS+1
TEST    AH,C2
JNZ     BIG_ANGLE
```

```
        TEST    AH,C1           ; Determine if PI/4 subtract needed
        JZ      DO_R            ; If zero, then no subtract
        FSUBRP  ST1,ST0         ; A=PI/4-R       ;        ?
        JMP     SHORT DO_FPTAN
DO_R:
        FXCH                    ;     PI/4       ;        R
        FCOMP                   ;      R         ;        ?
DO_FPTAN:
        FPTAN                   ;      OPP       ;        ADJ where OPP/ADJ=TAN(A)

;------ Determine if sin or cos required

        TEST    AH,C3 OR C1     ; Look at both
        JPE     DO_SINE
        FXCH                    ;      ADJ       ;        OPP
DO_SINE:                        ;       D        ;        N

;------ Calculate N/SQR(N**2 + D**2)

        FMUL    ST0             ;     D**2       ;        N
        FXCH    ST1             ;      N         ;       D**2
        FLD     ST0             ;      N         ;        N        ;      D**2
        FMUL    ST0             ;     N**2       ;        N        ;      D**2
        FADD    ST2             ; N**2+D**2      ;        N        ;      D**2
        FSQRT                   ;SQR(N2+D2)      ;        N        ;      D**2
        FDIVRP  ST1             ;  SIN(X)        ;       D**2
        FXCH    ST1             ;    D**2        ;       SIN(X)    ;
        FCOMP                   ;   SIN(X)
        TEST    AH,C0
        JZ      SIGN_OK
        FCHS
SIGN_OK:
        CALL    FLOAT_ASCII
        INC     NUM_ANGLE
        CMP     NUM_ANGLE,13
        JA      RETURN_INST
        JMP     DO_AGAIN
RETURN_INST:
        RET
BIG_ANGLE:
        MOV     DX,OFFSET ERROR_MSG
        MOV     AH,9
        INT     21H
        RET
SIN     ENDP
CODE    ENDS
        END     SIN
```

Figure 7.27(a) SIN routine

```
A>SIN
 4.79425539E-001
 8.41470985E-001
 9.97494987E-001
 9.09297427E-001
 5.98472144E-001
 1.41120008E-001
-3.50783228E-001
-7.56802495E-001
-9.77530118E-001
-9.58924275E-001
-7.05540326E-001
-2.79415498E-001
 2.15119988E-001
```

Figure 7.27(b) Display from SIN routine

Abbildung 7.27 Sinusberechnung. (a) SIN-Routine; (b) Ausgabe der SIN-Routine

Der erste Teil unseres Programms initialisiert dieses als .EXE-Datei. Danach lädt der 8087 die beiden Integerwerte und dividiert sie, um so für die weitere Verarbeitung den Winkelwert festzulegen. Dabei haben wir ein schönes Beispiel für die Verwendung von zwei Integerwerten, um eine Gleitpunktzahl zu bilden (in unserem Fall 1/2), eine Aufgabe, die der Assembler selbst nicht erfüllen kann.

Wie wir aus der Trigonometrie wissen, ist der Sinus eine Funktion, die sich wiederholt. Das heißt, die Funktion erzeugt gleiche Ergebnisse für Eingabewerte, die um jeweils genau 2*PI differieren. Deshalb ist die erste Aufgabe unseres Sinusprogramms, den Eingabewert des Winkels auf ein Maß zu reduzieren, das sich im Bereich

$$0 \leqq X < 2*PI$$

befindet.

Der Befehl FPTAN erfordert jedoch, daß sich der Winkel im Bereich

$$0 \leqq X < PI/4$$

befindet.

Dies bedeutet, daß selbst wenn der Winkel kleiner als 2*PI ist, wir ihn noch weiter reduzieren müssen, um den Bedingungen des Befehls FPTAN zu genügen. Glücklicherweise können wir selbst dann, wenn der Wert des Winkels auf weniger als PI/4 reduziert ist, immer noch korrekt mit trigonometrischen Funktionen arbeiten. Um dies durchzuführen, müssen wir allerdings wissen, wo in dem Bereich zwischen 0 und 2*PI sich der originale Winkel befand.

Der Befehl FPREM führt dies für uns aus. Dieser Befehl übergibt uns nämlich nicht nur den Divisionsrest, sondern er übermittelt uns auch die drei niederwertigen Bits des Quotienten, der sich während der Feststellung des Divisionsrestes ergab. Diese drei Bits werden im Statuswort gespeichert. Das bedeutet, daß wir den Oktanden, in den der Winkel fällt, selbst dann noch feststellen können, wenn wir den Originalwinkel auf ein Achtel seines ursprünglichen Werts reduziert haben. Nun können wir uns etwas mit der Trigonometrie befassen und die passende Formel für die Berechnung des Sinus entwickeln. In Abbildung 7.28 sehen wir den Zusammenhang zwischen dem originalen Oktanden und der Methode zur Berechnung des Sinus des gegebenen Winkels. Wir gehen dabei davon aus, daß der Wert R der Rest des ursprünglichen Winkels ist, nachdem dieser auf weniger als PI/4 reduziert wurde. Die Oktandenwerte sind dabei die Werte, die nach Ausführung des Befehls FPREM in den Bits C3-C1-C0 stehen.

Mit dieser Tabelle können wir außerdem die korrekte Formel bestimmen, die wir auf die einzelnen Schritte innerhalb unseres Programms anwenden müssen. Nachdem der Winkel im Bogenmaß bekannt ist, wählt unser Programm die Konstante PI und dividiert sie durch 4, um danach den Befehl FPREM auszuführen. An diesem Punkt müssen wir den Wert des Statusworts betrachten. Ist die Berechnung noch nicht vollständig ausgeführt, bedeutet dies, daß der Eingabewinkel größer als 2^{64} war. Da dieser Wert aber so weit über den passenden Werten der trigonometrischen Funktionen liegt, können wir ihn vernachlässigen, da er außerhalb jeder vernünftigen Verarbeitung liegt. Dies wird mit den Werten, die wir für unser Beispiel gewählt haben, nicht geschehen, wir weisen aber auf diesen Punkt hin.

	Oktand				
C0	C3	C1	Bereich		SIN(X) =:
0	0	0	0	PI/4	SIN(R)
0	0	1	PI/4	PI/2	COS(PI/4-R)
0	1	0	PI/2	3*PI/4	COS(R)
0	1	1	3*PI/4	PI	SIN(PI/4-R)
1	0	0	PI	5*PI/4	− SIN(R)
1	0	1	5*PI/4	3*PI/2	− COS(PI/4-R)
1	1	0	3*PI/2	7*PI/4	− COS(R)
1	1	1	7*PI/4	2*PI	− SIN(PI/4-R)

(R ist der Rest, $0 < R < PI/4$)

Abbildung 7.28 SIN(X) in den acht Oktanden

Unsere Routine überprüft nun im Statusregister das Bit C1, um festzustellen, ob es den Divisionsrest R weiter verwenden, oder ob es R von PI/4 subtrahieren soll. Befindet sich dabei der Wert PI/4 immer noch in einem der Register, so ist dieser Schritt verhältnismäßig einfach. Wird die Subtraktion dagegen nicht benötigt, so löscht der Befehl FCOMP den nicht benötigten Wert PI/4 aus dem Stack.

Der Befehl FPTAN ermittelt sodann den partiellen Tangens. Das Ergebnis wird angegeben als OPP/ADJ (für opposite divided by adjacent), was dem Tangens des Winkels R oder PI/4-R entspricht, in Abhängigkeit davon, was gerade ausgewählt war. Mit diesen beiden Werten können wir nun den Sinus oder Cosinus unseres Winkels bestimmen. So können wir beispielsweise, wenn OPP/ADJ gegeben ist, den Sinus nach der folgenden Formel berechnen:

SIN(X) = OPP/SQR(OPP2 + ADJ**2), wobei TAN(X) = OPP/ADJ**

Wollen wir den Cosinus berechnen, so vertauschen wir die beiden Operanden. Um zu entscheiden, ob sowohl Sinus oder Cosinus korrekt sind, sehen wir uns den gesicherten Oktanden an und testen außerdem die Werte C3 und C1 im Statuswort. Der Testbefehl isoliert dabei diese Werte und mit dem Befehl JPE können wir im Programm verzweigen, wenn beide Werte 0 oder beide Werte 1 waren. In diesem Falle berechnen wir den Sinus. Waren die beiden Werte verschieden, so berechnen wir den Cosinus, indem wir die Register OPP und ADJ im Registerstack vertauschen.

Die nächste Folge von Befehlen für den 8087 berechnet den Sinus (oder Cosinus) aus dem Wert für den partiellen Tangens. Der einzige Schritt, den wir dabei noch durchführen müssen, ist die Bestimmung des Vorzeichens des Ergebnisses. Für den Sinus wird das Ergebnis negativ, wenn sich der Winkel im vierten bis siebenten Oktanden befand. Ein Test des Bits C0 im Statuswort läßt uns das korrekte Vorzeichen für das Ergebnis feststellen. Die in Abbildung 7.25 gezeigte Routine FLOAT_ ASCII gibt die Ergebnisse sodann als Gleitpunktzahlen aus. Danach verzweigt die Schleifensteuerung zurück zum Anfang, wenn wir noch nicht alle möglichen Oktanden abgearbeitet haben. Der letzte Teil von Abbildung 7.27 zeigt den Output unseres Programms.

Fehlersuche mit dem 8087

Bevor wir die Diskussion des 8087 verlassen, sollten wir noch ein paar Worte über die Fehlersuche in Programmen für den 8087 verlieren. Das Problem, dem wir bei der Anwendung des Prozessors gegenüberstehen, ist die Tatsache, daß das DEBUG-Programm unter DOS den 8087 nicht unterstützt. Das bedeutet, daß beim Auftreten eines Breakpoints im DEBUG-Programm die ausgegebenen Registerwerte nicht die Register des 8087 mitenthalten. Dies macht es für uns sehr schwierig, den Ablauf eines Programms zu verfolgen, bei dem die Register des 8087 modifiziert werden.

Wir bieten allerdings eine Methode an, die verwendet werden kann, um Programme für den 8087 mit Hilfe des DOS DEBUG-Programms zu testen. Diese Methode wird zwar nicht die bestmögliche für alle Fälle sein, sie funktioniert aber zumindest für die Beispielprogramme in diesem Kapitel.

Das größte Hindernis bei der Fehlersuche ist die Unfähigkeit des DEBUG-Programms, die Inhalte des Registerstacks des 8087 auszugeben. Ohne nun lange das DEBUG-Programm verändern zu müssen, vermittelt Ihnen unsere Methode die nötigen Informationen, um die Fehlersuche innerhalb eines 8087-Programms zu ermöglichen. Die Voraussetzung dazu ist, daß Sie Ihr Programm als eigenständiges Programm schreiben, also entweder als .EXE-Datei oder als .COM-Datei. Selbst wenn Sie ein Unterprogramm schreiben, sollten Sie es zuerst als Hauptprogramm laufen lassen. Einer der ersten Befehle in diesem Programm ist der Befehl FINIT, der den 8087 in seinen Grundzustand nach dem Stromeinschalten versetzt. Dies ist notwendig, um zu ermöglichen, daß wir unser Programm immer wieder von Anfang an starten. Die hier gezeigte Methode erlaubt es nämlich nicht, den Befehlsfluß anzuhalten, die Register des 8087 zu untersuchen und dann an diesem Punkt wieder aufzusetzen. Stattdessen stützt sich unsere Methode auf die Möglichkeit, nach jedem Anhalten des Programms wieder ganz von vorne zu beginnen.

Sie sollten außerdem alle an die Routine übermittelten Parameter als Speicherstellen im Programm definieren. Das Programm sollte dann nach Ausführung des Befehls FINIT alle diese Werte in die passenden Register laden. Sie können diese Methode selbst dann verwenden, wenn Ihr Programm mit Werten arbeitet, die auf dem Registerstack übergeben werden. Das erste Austesten unseres Programms führen wir nun mit diesen im Speicher befindlichen Parametern durch. Haben wir solchermaßen die arithmetische Logik unseres Programms auf Fehlerfreiheit überprüft, so können wir unser Programm dahingehend verändern, daß es im weiteren die Parameter auch aus dem Registerstack akzeptiert.

Das Ziel unserer Modifikationen ist dabei, dem Programm zu ermöglichen, ohne Interventionen von außen abzulaufen. Das heißt, wir können unser Programm immer von Anfang an starten und es dann bis zu einem bestimmten Befehl laufen lassen. Ein Neustart von Anfang an führt das Programm wieder in genau der gleichen Weise aus. Wir benötigen diese Fähigkeit, da die von uns verwendete Methode zur Ausgabe der Registerwerte den Inhalt des Stacks des 8087 zerstört. Haben wir auf diese Weise den Stack einmal modifiziert, so können wir nicht einfach das Programm wieder fortsetzen. Wir müssen also wiederum von Anfang an beginnen und das Programm bis zu einer anderen Stelle laufen lassen. Unsere Vorschläge gestatten es Ihnen, das Programm solchermaßen aufzubauen. Die beiden letzten Beispielprogramme, die quadratische Gleichung und die Sinusfunktion, sind beide in dieser Weise gestaltet. Die Parameter befinden sich also in Speicherbereichen, und die Programme beginnen jeweils mit dem Befehl FINIT.

Der nächste Schritt für eine erfolgreiche Fehlersuche benötigt eine spezielle Befehlsfolge an einer ganz bestimmten Stelle in Ihrem Programm. In unserem Beispiel verwendeten wir dazu die Stelle 200H, da keines der Beispielprogramme länger als 500 Bytes ist. Diese Befehlsfolge dient nur zu Testzwecken in Ihrem Programm und Sie sollten sie deshalb entfernen, wenn Sie das Programm in seine endgültige Fassung bringen. In Abbildung 7.29 finden Sie die besprochene Befehlsfolge. Wie Sie sehen, ist diese sehr kurz und besteht nur aus drei Befehlen und zwei Datenbereichen. Der erste Datenbereich ist dabei eine Konstante, in unserem Beispiel 10^6 oder eine Million. Die Auswahl dieses besonderen Wertes bleibt

jedoch ganz Ihnen überlassen. Bearbeitet Ihr Programm nämlich Zahlen, die wesentlich kleiner als 10^{-6} oder größer als 10^{12} sind, so könnte eine andere Konstante angemessener erscheinen.

Der Zweck unseres Programmfragmentes ist es nun, die Spitze des Stacks in eine Zahl zu verwandeln, die wir betrachten können. Zu diesem Zweck wird der an der Stackspitze enthaltene Wert mit einer Zahl multipliziert, die eine Menge Nullen enthält. Die Folge davon ist ein Schieben des Dezimalpunkts nach rechts. Würde im Beispiel der Stack 1/2 enthalten, so erzeugt die Multiplikation ein Ergebnis von 500.000.

Haben wir nun auf diese Weise die Zahl aus einem gebrochenen Wert in eine große Integerzahl verwandelt, können wir sie mit dem Befehl FBSTP als gepackte BCD-Zahl speichern. Der Speicherbereich dafür ist in unserem Programm vorgesehen. Der Befehl INT 3 übergibt die Steuerung sodann an die DEBUG-Routine. Wir können nun das Display-Kommando des DEBUG-Programms verwenden, um uns diese zehn Bytes anzusehen, die vorher vom Befehl FBSTP gespeichert wurden. Natürlich müssen Sie in diesem Fall die Werte von hinten lesen, denn dies ist die Art, in der der 8087 BCD-Zahlen speichert. Sie müssen außerdem berücksichtigen, daß der Dezimalpunkt durch die erfolgte Multiplikation verschoben wurde.

```
TEN6        DD        1000000
            ORG       200H
BCD_TEMP              DT        ?
            ORG       210H
            FIMUL     TEN6
            FBSTP     BCD_TEMP
            INT       3
```

Abbildung 7.29 DEBUG-Routine für den Arithmetikprozessor

Die Fehlersuche in einem 8087-Programm läuft nun etwa folgendermaßen ab: Nachdem Sie festgestellt haben, daß Ihr Programm nicht ganz korrekt läuft, suchen Sie eine geeignete Stelle für eine Programmunterbrechung. Die Verwendung des Deassemblierungskommandos hat dabei keinen großen Nutzen, denn alle Befehle für den 8087 werden als Formen des Befehls ESC wiedergegeben. Es ist also wichtig, daß Sie über eine Programmliste verfügen.

Wir führen nun unser Programm vom Anfang bis zu dem gewünschten Unterbrechungspunkt aus. Dies ist auch der Grund, warum wir unser Programm so entwikkelt haben, daß es immer wieder von Anfang an gestartet werden kann. Jedesmal, wenn wir nämlich einen neuen Unterbrechungspunkt setzen, müssen wir unser Programm wieder ganz von vorne beginnen.

Erreichen wir nun in unserem Programm einen Unterbrechungspunkt, so geht die Steuerung zurück an das DEBUG-Programm. Jetzt können wir den speziellen Befehlsteil ausführen, den wir bereits eingeschlossen haben. Der Befehl INT 3 die-

ses Programmstücks gibt die Steuerung dann wieder zurück an das DEBUG-Programm, so daß wir einen Blick auf den BCD-Datenbereich werfen können und damit sehen, welche Werte sich im Stack befanden, als die Unterbrechung ausgeführt wurde. Da wir zur Ausgabe der Daten den Befehl FBSTP verwendet haben, befindet sich der Wert, den wir dabei im Speicher ablegten, nicht mehr im Stack. Wir können nun unser Testprogramm-Stück ein weiteres Mal laufen lassen, um den zweiten Wert im Stack, ST1, auszugeben. Dies können wir sooft wiederholen, wie wir wollen. Erhalten wir auf diese Weise eine BCD-Zahl mit dem Wert 0FFH sowohl in der höchstwertigen Ziffernstelle als auch im Vorzeichen, so wissen wir, daß wir einen leeren Eintrag aus dem Stack gelesen haben. Jetzt können wir einen neuen Unterbrechungspunkt im Programm setzen und das Programm wieder von vorne beginnen lassen. Auf diese Weise können wir uns schrittweise durch unser Programm tasten, bis wir schließlich die Problemzone finden. Haben wir so den Fehler entdeckt, so können wir entweder den Code an Ort und Stelle verändern (das Problem „patchen") oder aber zurück ins Betriebssystem kehren, um das Programm erneut zu editieren und dann zu reassemblieren. Läuft unser Programm schließlich korrekt und wir benötigen die DEBUG-Routine nicht länger, können wir unseren Test-Programmteil aus dem eigentlichen Programm entfernen. Wir können das Programm dann auch dahingehend verändern, daß es beispielsweise Parameter aus den Stackregistern übernimmt, anstelle Speichervariablen zu verwenden.

8 Der IBM Personal Computer

Dieses Kapitel erläutert die Hardware des IBM Personal Computers. Nachdem sich nämlich dieses Buch hauptsächlich mit der Assemblerprogrammierung mit dem IBM Personal Computer befaßt, ist es angemessen, auch die darunter liegende Hardware zu besprechen. Dieses Kapitel ist dabei nicht für Ingenieure oder Techniker bestimmt. Es ist eher für solche Leute gedacht, die Assemblerprogramme schreiben, die auf dem IBM PC ablaufen sollen.

Wir wir bereits besprochen haben, ist die Assemblersprache nicht in jedem Fall die geeignete Programmiersprache. Doch das Programmieren direkt auf Maschinenebene erlaubt es dem Programmierer, eine verhaltnismäßig große Kontrolle über die Maschine auszuüben. In großen Projekten macht es der überwältigende Aufwand für Details allerdings schwierig, sich bei Verwendung der Assemblersprache auf die eigentlichen Ziele, die es zu lösen gibt, zu konzentrieren. Es ist deshalb das beste, die Assemblersprache immer dann zu verwenden, wenn sie wirklich benötigt wird.

Einer der Gründe für die Verwendung der Assemblersprache könnte sein, daß Sie beispielsweise daraus Nutzen ziehen wollen, direkte Kontrolle über die Hardware auszuüben. Um diese Aufgabe korrekt zu erfüllen, müßten Sie wissen, was die Hardware ist und was sie tut. Und dies ist auch der Grund für dieses Kapitel. Die Information im folgenden Kapitel ist also für den Programmierer, nicht für den Ingenieur bestimmt. Wir werden dabei die einzelnen Teile der Hardware besprechen und auch, wie wir sie programmieren können.

Die Information in diesem Kapitel ist dabei eine Ergänzung zur Beschreibung im Personal Computer Technical Reference Manual. Für einige spezielle Aspekte der Programmierung der Hardware sollten Sie sich also auf dieses Technical Reference Manual beziehen. Auch aus den Spezifikationsblättern für die einzelnen Hardwaregeräte können Sie zusätzliche Informationen erhalten. Auf diese Daten werden wir in diesem Kapitel nicht eingehen. Wo es allerdings notwendig ist, werden wir im Text einige der Hardwaredaten wiedergeben, um bestimmte Programmvorgänge zu erklären. Natürlich werden wir auch Beispiele bringen, die das Arbeiten der Hardware erläutern.

Eine Besprechung des Basic Input/Output-Systems (BIOS), das sich im Read-only Memory des Rechners (ROM) befindet, werden wir auf das nächste Kapitel verschieben. Diese BIOS-Routinen erlauben uns Zugriff auf die Ein/Ausgabegeräte des IBM PC auf direkter Geräteebene. In diesem Kapitel werden wir nun erklären, was die Hardware tut. Im nächsten Kapitel erläutern wir dann im einzelnen, was wir mit der Hardware tun können, und im letzten Kapitel werden wir Ihnen schließlich helfen, mit der Hardware einige Dinge zu tun, die im ursprünglichen ROM BIOS nicht enthalten sind.

Systemhardware

Wir werden nun in einzelnen Abschnitten die wichtigsten Teile des IBM PC besprechen. Im folgenden Abschnitt behandeln wir die Standardkomponenten der Hardware – die Teile der Hardware, die sich auf der Systemplatine befinden. In weiteren Abschnitten werden wir uns dann mit den einzelnen Ein/Ausgabe-Adapterkarten befassen, die Sie nach Wunsch in Ihrem System installieren können.

Der Hauptprozessor des IBM PC ist der Intel 8088. Dies ist natürlich der Prozessor, den wir in den ersten Teilen dieses Buchs lang und breit besprochen haben. Zum Glück gibt es nur noch wenig mehr, was wir über diesen Prozessor sagen könnten. Neben dem 8088 befindet sich auf der Systemplatine ein leerer Stecksockel, der den numerischen Datenprozessor Intel 8087 aufnehmen kann. Diesen Arithmetikprozessor haben wir bereits im Kapitel 7 besprochen, er sollte Ihnen also bekannt sein.

Die restlichen Komponenten auf der Systemplatine führen Funktionen aus, die aus dem Mikroprozessor einen Computer machen. Dazu befinden sich auf der Systemplatine bis zu 64K Bytes Schreib-/Lesespeicher und 40K Lesespeicher (ROM). Dieser ROM enthält sowohl den Basic-Interpreter als auch das ROM BIOS, das wir im nächsten Kapitel besprechen werden.

Es gibt eine Menge von Komponenten auf der Systemplatine, die für die Arbeit des IBM PC von entscheidender Bedeutung sind. Wir werden uns nur auf die Komponenten konzentrieren, die programmierbar und außerdem für uns von Nutzen sind. Auf der Systemplatine zählen dazu der 8255 (ein programmierbares peripheres Interface), der 8253 (Zeitgeber, Zähler), der 8259 (Unterbrechungscontroller) und der 8237 (Speichersteuerung). Die restlichen Komponenten haben reine Hardwarefunktionen, die nicht durch Programmieren modifiziert werden können. Natürlich werden wir jetzt diese Teile nicht Stück für Stück und nacheinander untersuchen. Dazu sollten Sie den Intel Bauelemente-Datenkatalog verwenden oder irgendein anderes Referenzmaterial. Wir werden uns stattdessen mit den Ein/Ausgabefunktionen befassen, die auf der Systemplatine des IBM PC implementiert sind. Wenn wir die Steuerung dieser Geräte übernehmen wollen, tun wir dies unter Benutzung der bereits erwähnten Komponenten.

Lautsprecher

Der IBM PC verfügt über einen eingebauten kleinen Lautsprecher. In einem Programm können wir die Töne steuern, die von diesem Lautsprecher erzeugt werden. Um dies zu ermöglichen, müssen wir einige der Outputbits des 8255 und außerdem den Tongenerator im 8253 steuern.

In Abbildung 8.1 sehen wir ein Programm, das den Lautsprecher auf zwei verschiedene Methoden anspricht. Die erste Methode, mit dem Label DIRECT in der Programmliste bezeichnet, steuert den Lautsprecher direkt an. Bit 1 des Outputports 61H ist dabei direkt mit dem Lautsprecher verbunden. Jedesmal, wenn wir nun den

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.1  Speaker Control

 1                                              PAGE    ,132
 2                                              TITLE   Figure 8.1  Speaker Control
 3      0000                            STACK   SEGMENT STACK
 4      0000        40 [                        DW      64 DUP(?)
 5                          ????
 6                                  ]
 7
 8      0080                            STACK   ENDS
 9
10      0000                            CODE    SEGMENT
11                                              ASSUME  CS:CODE
12
13      0000                            SPEAKER PROC    FAR
14      0000    1E                              PUSH    DS              ; Set return address
15      0001    B8 0000                         MOV     AX,0
16      0004    50                              PUSH    AX
17
18                                      ;----- Direct Control of Speaker
19
20      0005    2B C9                           SUB     CX,CX           ; Loop count
21      0007    E4 61                           IN      AL,61H
22      0009    24 FE                           AND     AL,0FEH
23      000B    E6 61                           OUT     61H,AL          ; Set bit 0 to 0 for direct control
24      000D                            DIRECT:
25      000D    0C 02                           OR      AL,2
26      000F    E6 61                           OUT     61H,AL          ; Turn on speaker
27      0011    24 FD                           AND     AL,0FDH
28      0013    E6 61                           OUT     61H,AL          ; Turn off speaker
29      0015    E2 F6                           LOOP    DIRECT
30
31                                      ;----- Tone Control of Speaker
32
33      0017    B0 B6                           MOV     AL,10110110B
34      0019    E6 43                           OUT     43H,AL          ; Set channel 2 for correct mode
35      001B    B8 03E8                         MOV     AX,1000         ; Select tone
36      001E    E6 42                           OUT     42H,AL
37      0020    8A C4                           MOV     AL,AH
38      0022    E6 42                           OUT     42H,AL          ; Output tone to timer
39
40      0024    E4 61                           IN      AL,61H
41      0026    8A E0                           MOV     AH,AL           ; Save original in AH
42      0028    0C 03                           OR      AL,3
43      002A    E6 61                           OUT     61H,AL          ; Select tone control
44
45      002C    2B C9                           SUB     CX,CX
46      002E                            KILL_TIME:
47      002E    E2 FE                           LOOP    KILL_TIME       ; Waiting loop
48
49      0030    8A C4                           MOV     AL,AH
50      0032    E6 61                           OUT     61H,AL          ; Turn speaker off
51
52      0034    CB                              RET
53      0035                            SPEAKER ENDP
54      0035                            CODE    ENDS
55                                              END
```

Abbildung 8.1 Lautsprechersteuerung

Wert dieses Bits verändern, bewegt sich die Membrane des Lautsprechers entweder nach innen oder nach außen. Wenn wir nun den Wert dieses Bits sehr schnell verändern, erzeugt unser Programm einen Ton. Im ersten Teil von Abbildung 8.1 sehen wir dies, wobei der Wert von Bit 1 jeweils verändert und dadurch ein sehr hoher, pfeifender Ton erzeugt wird. Die Geschwindigkeit, mit der unser Programm das Bit 1 ändert, bestimmt dabei die Frequenz des erzeugten Tons.

Um direkte Kontrolle über den Lautsprecher zu erlangen, müßten wir zuerst den Outputport des 8255 auf der Systemplatine manipulieren. Der 8255 (Programmable Peripheral Interface — PPI) enthält insgesamt drei Input- oder Outputports. Der IBM PC initialisiert den 8255 in einer Weise, daß zwei Inputports — Port 60H und 62H — und ein Ausgabeport — 61H — vorgesehen sind. Port 60H dient dabei vorzugsweise zum Lesen von Tastaturdaten. Wir können den Port 60H allerdings auch dazu verwenden, bestimmte Schalterstellungen auf der Systemplatine zu lesen. Allerdings werden die Schalter normalerweise nur ein einziges Mal gelesen, und zwar während der Einschaltphase des Systems. Das ROM BIOS legt diese Werte dann für spätere Verwendung im Speicher ab. Für unsere Zwecke behandelt Port

60H ausschließlich Tastaturdaten. Ein Inputport hat dabei eine wichtige Aufgabe. Er dient nämlich als Puffer zwischen dem Prozessor und dem Ein/Ausgabegerät. Der Port präsentiert die Daten dabei dem Prozessor nur, wenn der Prozessor diese verlangt — während eines IN-Befehls. Zu allen anderen Zeitpunkten hält der Inputport die Daten zwar bereit, gestattet aber nicht, daß sie in irgendeiner Weise Einfluß auf den Prozessor nehmen.

Port 62H, der andere Inputport des 8255, behandelt eine Vielzahl von anderen Inputs. Vier der Inputbits sind dabei direkt an einen Schalter angeschlossen, der die Größe des Speichers angibt, der im Ein/Ausgabekanal des Systems zur Verfügung steht. Die restlichen vier Bits haben jeweils eine eigene Bedeutung. Zwei von ihnen zeigen bestimmte Systemfehler an. Die Routine NMI (Non Maskable Interrupt) verwendet diese Bits, um die Ursache eines Systemfehlers festzustellen. Bit 5 des Ports 62H sieht einen Feedback-Mechanismus für einen der Zeitgeber-/Zählerkanäle vor. Dieses Bit enthält dabei den gerade aktuellen Output des Kanals 2 des 8253. Bit 4 des Ports 62H enthält den aktuellen Status des Kassetteninputs. Der IBM PC verfügt nämlich über einen Kassettenanschluß an der Rückseite neben dem Tastaturstecker. Lesen wir nun Daten von der Kassette, so wird dieses Inputbit verwendet, um den aktuellen, von der Kassette gelesenen Datenwert zu bestimmen.

Port 61H ist der Ausgabeport des 8255 im IBM PC. Ein Ausgabeport beinhaltet dabei für einige Zeit den vom Programm ausgegebenen Datenwert. Würden die Werte von der Hardware nicht für eine ganz kurze Zeit gespeichert, würden sie innerhalb von Mikrosekunden verschwinden. Ein Datenregister hält dabei den Wert solange konstant, bis er vom Programm wieder geändert wird. Das bedeutet, wenn wir einen Ausgabewert an den Lautsprecher schicken, so bleibt dieser Wert solange gesetzt, bis er vom Programm wieder verändert wird.

Abbildung 8.2 zeigt die Bitwerte für den Port 61H. Diese Daten sind aus dem technischen Referenzmanual entnommen.

Bit	Bedeutung
0	Zeitgeber 2 (Lautsprechersteuerung)
1	direkte Lautsprechersteuerung
2	Multiplexen Port 62H
3	Kassettenmotorsteuerung
4	Parity-Prüfung auf Systemplatine ein
5	Parity-Prüfung in I/O-Kanälen ein
6	Tastatur-Zeitgeber
7	Tastatur löschen / Multiplexen Port 60H

Abbildung 8.2 Bedeutung der Bits von Port 61H

Für unsere Diskussion der Steuerung des Lautsprechers spielen nur die Bits 0 und 1 eine Rolle. Von den anderen Bits haben nur Bit 3 — Steuerung des Kassettenmotors — und Bit 7 — Rücksetzen der Tastatur — für unsere Programme noch einige Bedeutung. Alle anderen Bits sind für Initialisierungs- und Diagnosezwecke gedacht. Um ihre Bedeutung voll zu verstehen, müßten wir tief in die Schaltpläne des Systems einsteigen.

Kehren wir nun wieder zurück zu unserer Besprechung der Steuerung des Lautsprechers, wobei wir sehen, daß die Bits 0 und 1 für die direkte Steuerung verwendet werden. Wie Abbildung 8.1 zeigt, wird durch das Setzen des Bits 0 auf den Wert 0 die direkte Steuerung des Lautsprechers initialisiert. Dadurch wird der Tongenerator des 8253 ausgeschaltet. Diese Methode werden wir in der zweiten Hälfte unseres Programms weiter verfolgen.

Halten wir außerdem fest, wie unser Programm das Bit 0 rücksetzt. Der Befehl OUT setzt alle acht Bits des Ports 61H. Es gibt also keine Möglichkeit, nur das Bit 0 zu verändern und alle anderen unberührt zu lassen. Da wir aber in unserem Programm nur das Bit 0 modifizieren wollen, muß der gesamte aktuell anliegende Wert vom Port gelesen werden. Glücklicherweise erlaubt es uns der 8255, auch einen Outputport direkt zu lesen.

Die gezeigte Befehlsfolge liest den gerade anliegenden Wert direkt vom Outputport.

```
IN      AL,61H
AND     AL,0FEH
OUT     61H,AL
```

Der AND-Befehl löscht dann das niedrigstwertige Bit, und der OUT-Befehl sendet diesen Wert wiederum als Kontrollbyte zurück. Hätten wir in unserem Programm den Wert 0 direkt auf den Port ausgegeben, hätte zwar der Lautsprecher korrekt gearbeitet, die Tastatur jedoch wäre blockiert gewesen. Das bedeutet, daß wir bei der Bearbeitung von bitsignifikanten Werten an einem Outputport immer sicherstellen müssen, daß unser Programm nicht auch die anderen Bits verändert, außer wir wollen dies absichtlich tun.

Der Rest des ersten Programms in Abbildung 8.1 verändert den Wert von Bit 1 des Outputports. Der ursprüngliche Wert des Ports 61H befindet sich dabei bereits im Register AL, so daß unser Programm nicht bei jedem Durchlauf durch die Schleife den Wert erneut lesen muß. Wir verwenden dabei das CX-Register, um die Schleife 64.000 Mal zu durchlaufen. Führen wir dieses Programm aus, so könnten Sie Schwierigkeiten haben, den ersten Ton zu hören, den unser Programm erzeugt. In diesem Fall sollten Sie versuchen, einige zusätzliche Leerbefehle (NOP) in die DIRECT-Schleife einzufügen. Der Erfolg ist eine verlängerte Laufzeit und damit ein niedrigerer Ton.

Der zweite Teil von Abbildung 8.1 behandelt die Verwendung des 8253 (Zähler/Zeitgeber), um einen Ton zu erzeugen. Bevor wir nun weitergehen, wollen wir zuerst die Arbeitsweise des 8253 besprechen, um zu sehen, wie er im System verwendet wird. Der Intel 8253 verfügt über drei 16-Bit Zähler, die im System für Zeitgeberzwecke oder zum Zählen verwendet werden können. Dazu laden wir einen beliebigen 16-Bit Wert in einen der Zähler. Durch jeden Taktimpuls wird der Zähler dann um jeweils 1 erniedrigt. Die Taktfrequenz für jeden der drei Zähler beträgt dabei 1.19 MHz. Das bedeutet, daß der Zähler alle 840 Nanosekunden um jeweils 1 erniedrigt wird. Jeder der drei Kanäle verfügt über einen eigenen Ausgang. Das Signal der Ausgangsleitung ändert sich dabei jedesmal, wenn der Zähler Null erreicht. Durch zusätzliche Steueranweisungen kann der 8253 bestimmen, auf welche Weise der Zählvorgang abläuft.

Die Ausgänge dieser Zeitgeber- bzw. Zählerkanäle sind mit verschiedenen Geräten auf der Systemplatine verbunden. Kanal 0 ist beispielsweise mit dem 8259 (Unterbrechungscontroller) verbunden. Er wird vom System für eine Zeitgeber(Uhr)-Funktion verwendet. Kanal 1 ist mit der 8237 DMA-Steuerung verbunden, wobei zu bemerken ist, daß wir diesen Kanal möglichst nie verwenden sollten. Das Verändern der Werte in diesem Zähler könnte nämlich sehr leicht dazu führen, daß Ihr Programm zerstört wird, zusammen mit all den anderen Daten, die sich gerade im Speicher befinden. Kanal 2 ist zur Tonerzeugung mit dem Lautsprecher verbunden.

Wir werden später noch einmal auf den Kanal 0 des 8253 zurückkommen. Kanal 2 ist als Tongenerator für den Lautsprecher gedacht. Um diesen Zeitgeberkanal zu initialisieren, sendet unser Programm den Wert 0B6H an den Port 43H, den Steuerport des 8253. Dadurch wird bewirkt, daß der Kanal 2 des Zeitgebers bzw. Zählers als Frequenzteiler arbeitet. Der Zeitgeber dividiert also die Eingangsfrequenz – in unserem Fall 1.19 MHz – durch jeden beliebigen 16-Bit Wert, den unser Programm in das Kanal-2-Register geladen hat. Das Kanal-2-Register befindet sich an der Portadresse 42H (Kanal 0 hat die Portadresse 40H, und da Sie niemals den Kanal 1 modifizieren sollten, überlassen wir es Ihnen, die Adresse dieses Ports herauszufinden). Unser Programm lädt nun den Wert 1000 in dieses Kanalregister. Das bedeutet, daß wir eine Ausgabefrequenz von 1190 Hz hören werden. In der Tat hören wir allerdings nur eine Grundfrequenz von 1190 Hz und zusätzlich einige Obertöne, die durch den rechteckigen Puls des Zeitgebers erzeugt werden.

Halten wir dabei fest, daß 1000 eine 16-Bit Zahl ist, während der Port 42H nur 8 Bits breit ist. Der Modusbefehl, den wir an den Port 43H gaben, teilt dem 8253 mit, einen 16-Bit Wert in Form von zwei getrennten 8-Bit Werten zu erwarten. Dabei wird das niedrigwertige Byte zuerst übertragen, gefolgt vom höherwertigen Byte. Durch diesen zweistufigen Vorgang laden wir den korrekten Wert in das Kanalregister.

Im weiteren müßten wir nun den Steuerport 61H setzen, um unserem bereits erzeugten Ton zu erlauben, bis zum Lautsprecher vorzudringen. Dazu setzen wir die Bits 0 und 1 des Steuerports auf 1. Halten wir dabei fest, daß der ursprüngliche Wert des Kontrollports gespeichert und nach Ausführung der Operation auch wieder hergestellt wird. Dadurch wird der Lautsprecher nach dem gewünschten Ton wieder abgeschaltet. Sollte diese Methode nicht genügen – beispielsweise, weil unser Programm einen Ton während einer Zeit erzeugt, wo es nicht sicher ist, daß der Lautsprecher bereits abgeschaltet ist – können wir den Lautsprecher zwangsweise abschalten, indem wir das Bit 1 des Ports 61H auf 0 setzen.

Diese beiden gezeigten Methoden der Steuerung des Lautsprechers sind die einfachsten. Natürlich können Sie jede beliebige Kombination dieser Methoden ausprobieren, wenn Sie einige interessante Effekte erleben wollen. Wir könnten beispielsweise mit dem 8253 einen Ton erzeugen und diesen dann mit Bit 1 oder Bit 2 oder auch allen beiden des Ports 61H modulieren. Auch könnten wir den Wert des Frequenzteilers verändern, während der Lautsprecher eingeschaltet ist. Wir könnten auch das Programm in Abbildung 8.1 dahingehend verändern, daß der Wert, den das CX-Register nach Durchlauf einer jeden Schleife enthält, ausgegeben wird. Das würde dazu führen, daß sich der auf dem Lautsprecher ausgegebene Ton von einer sehr niedrigen Frequenz langsam zu einer sehr hohen fortbewegt. Mit diesen drei Kontrollwerten können wir eine Menge interessanter Effekte erzeugen.

Tastatur

Das nächste Ein/Ausgabegerät, das wir uns ansehen wollen, ist die Tastatur. Die Tastatur ist frei beweglich und mit dem System nur mit einem vieradrigen Spiralkabel verbunden. Obwohl die Tastatur selbst einen eigenen Mikroprozessor enthält, gibt es keinen geeigneten Weg, Programme für diesen zu schreiben. Wir sind also gezwungen, das Programmieren der Tastatur den Ingenieuren von IBM zu überlassen. Dafür können wir die Informationen verarbeiten, die uns die Tastatur sendet.

Der Tastaturschaltkreis auf der Systemplatine ist mit dem Interruptsystem verbunden. Sobald die Elektronik dabei einen Tastendruck registriert, wird dem System eine Unterbrechung signalisiert. Durch diese Unterbrechung wird die Steuerung an die Tastaturunterbrechungsroutine übertragen. Die Routine übernimmt die Daten von der Tastatur und sichert sie für einen späteren Gebrauch. Die Tastaturroutine behandelt auch Spezialfälle, so z.B. den Warmstart des Systems — System Reset (CTL-ALT-DEL) und Programmunterbrechungen (CTL-BREAK). Wir werden dies allerdings nicht jetzt, sondern erst im nächsten Kapitel besprechen, denn diese Unterbrechungen werden vom ROM BIOS abgehandelt. Wir sehen uns dafür an, wie wir die Hardware für die Tastatur steuern können.

Jedesmal, wenn die Tastatur einen Interrupt signalisiert, geht dieses Signal über den 8259 Interruptcontroller an den 8088. Der 8259 bearbeitet dabei praktisch alle Möglichkeiten des Unterbrechungssystems für den IBM PC.

Der 8259 kann bis zu acht verschiedene Geräte behandeln, die ihrerseits Interrupts erzeugen können. Im IBM PC sind dabei der Systemzeitgeber, die Tastatur, der Asynchronadapter, die Festplatte, die Diskette und der Drucker an die Unterbrechungsleitungen angeschlossen. Die restlichen Interruptebenen sind für andere Ein/Ausgabegeräte vorgesehen, die sich im System-Ein/Ausgabekanal befinden. Vom Design der Hardware her sind dabei jedem dieser Geräte, die eine Unterbrechung erzeugen können, eigene Interrupteingänge auf dem 8259 vorgesehen. Der Unterbrechungseingang, über den ein Gerät dabei angeschlossen ist, wird als Interruptebene für dieses Gerät bezeichnet. Wie wir gleich sehen werden, ordnet der 8259 die Unterbrechungen nach Kategorien ein, die diesen Ebenen entsprechen. In Abbildung 8.3 sehen wir die Unterbrechungsebenen für jedes der IBM-Geräte.

Wir können jedes der Geräte, die Interrupts erzeugen, einzeln aus- oder einschalten. So wie der 8088 ein Interruptflag hat, das es uns gestattet, Interrupts mittels der Befehle STI und CLI ein- oder auszuschalten, so tut dies auch der 8259. Der 8259 verfügt allerdings über acht Interruptflags, eines für jedes mögliche Gerät. Das Interruptmaskenregister (IMR) enthält diese acht Bits und befindet sich an der Portadresse 21H. Bit 7 entspricht dabei Interrupt 7, Bit 6 dem Interrupt 6 usw. Wenn wir eines dieser Bits auf 1 setzen, so kann das entsprechende Gerät keine Interrupts erzeugen. Setzen wir das entsprechende Bit auf 0, so läßt der Interruptcontroller die entsprechende Unterbrechung durch zum 8088. Natürlich muß, selbst wenn das IMR des 8259 einen bestimmten Interrupt zuläßt, auch das Interruptflag des 8088 eine solche Unterbrechung zulassen, bevor sie auftreten kann.

Der Interruptcontroller weist dabei jedem Gerät eine entsprechende Priorität zu, die von der jeweiligen Interruptebene abhängt. Diese Interruptebene wird durch die jeweilige Hardwareverbindung festgelegt und kann nicht durch Programmierung verändert werden. Interrupt 0 ist die höchste Priorität und Interrupt 7 die niedrigste. Versuchen nun zwei beliebige Geräte zu gleicher Zeit einen Interrupt zu erzeugen, so erhält das Gerät mit der höheren Priorität den Vorrang. Das Gerät mit der niederen Priorität wird solange angehalten, bis das System die Unterbrechung des höherwertigen Geräts abgearbeitet hat. Dieses Vorrangsystem wird von der internen Logik des 8259 abgehandelt, doch auch unser Programm muß an der Kontrolle der Interrupts mitwirken. Wir müssen dem 8259 nämlich mitteilen, wenn wir die Bearbeitung einer Unterbrechung beendet haben. Dieser Befehl, als „End of Interrupt" (EOI) bezeichnet, teilt dem 8259 mit, daß die Behandlung des Geräts mit der höheren Priorität beendet ist und daß nun das Gerät mit der niedrigeren Priorität seinerseits den Interrupt erzeugen kann.

Ebene	Gerät
0	Zeitgeber Kanal 0
1	Tastatur
2	–
3	Asynchrone Schnittstelle
4	zusätzl. asynchrone Schnittstelle
5	Festplatte
6	Diskette
7	Drucker

Abbildung 8.3 Interruptebenen des 8259

Der 8259 verhindert auch, daß ein Gerät mit niedrigerer Priorität den Prozessor unterbricht, solange das System gerade den Interrupt eines Geräts mit höherer Priorität bearbeitet. Der 8259 merkt sich dabei den gerade aktiven Interrupt und nimmt an, daß der Prozessor diesen Interrupt bearbeitet, solange er keinen EOI-Befehl erhalten hat. Der 8259 kann dies über zwei interne Register bewirken. Das erste Register identifiziert dabei die Geräte, die gerade einen Interrupt benötigen, und wird auch als Interrupt Request Register (IRR) bezeichnet. Ein zweites Register zeichnet die Interrupts auf, die gerade abgearbeitet werden und wird deshalb als In-Service-Register (ISR) bezeichnet. Haben wir nun in einer Interruptroutine den EOI-Befehl vergessen, so können nur noch Geräte mit höherer Priorität den Prozessor unterbrechen. Vergesen wir den EOI-Befehl nach einem Zeitgeberinterrupt, so kommt das System zum Stehen. Da der Zeitgeber nämlich die höchste Priorität hat, wären alle anderen Interrupts verhindert, solange wir nicht diese Interruptroutine korrekt abgeschlossen haben.

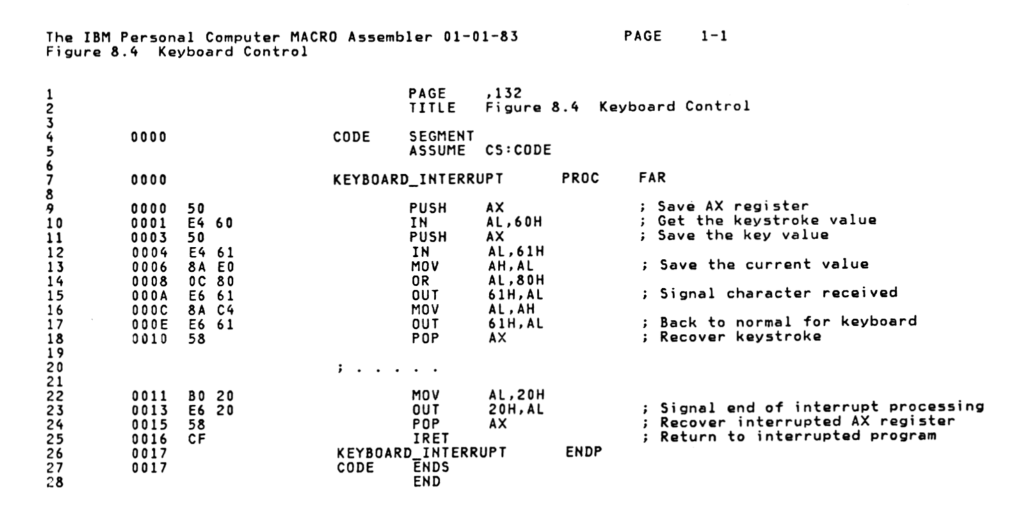
Der 8259 vektorisiert außerdem die Unterbrechungen. Dies bedeutet, daß der Controller den Prozessor veranlaßt, die jeweils korrekte Unterbrechungsroutine anzuspringen. Erhält der Controller nun eine Unterbrechung, so veranlaßt er den 8088, das Programm an einer Stelle fortzuführen, die über einen der 256 möglichen Interruptvektoren im ersten K-Byte des Speichers adressiert ist. Der IBM PC verwendet dabei für die acht Interruptebenen des 8259 die Interruptvektoren 8 bis 0FH. Tritt

also eine Tastaturunterbrechung (Interruptebene 1) auf, so beginnt unser Programm die Ausführung an der Stelle CS:IP, die durch den Inhalt des Doppelworts des Interruptvektors 9, also die Stelle 24H bis 27H, definiert ist.

Sehen wir uns nun einmal an, wie dies im IBM PC alles zusammenarbeitet. Als erstes initialisiert der Power-On Self-Test (POST) den 8259 mit den passenden Steuer- und Vektorinformationen. Zu diesem Zeitpunkt sind durch den POST der Zeitgeber, die Tastatur und die Disketteninterrupts im 8259 unterbunden. Nach Ausführung des POST wird das Interruptsystem mit dem Befehl STI eingeschaltet. Das System ist nun bereit, alle Interrupts anzunehmen.

Wir drücken jetzt beispielsweise eine Taste auf der Tastatur. Die Tastatur sendet nun das Zeichen an die Systemplatine, wo es in einem Register gespeichert wird und außerdem eine Unterbrechung der Ebene 1 erzeugt. Jetzt übernimmt der 8259 die Arbeit. Er setzt Bit 1 im IRR, um anzuzeigen, daß eine Unterbrechungsanforderung besteht. Sind weder Bit 0 noch Bit 1 des ISR gesetzt, ist also keine höherrangige Unterbrechung in Bearbeitung, so aktiviert der Controller die Unterbrechungsleitung des 8088. Ist der 8088 bereit, den Interrupt zu akzeptieren, so führt der Prozessor einen Unterbrechungsannahmezyklus aus. Der 8088 schiebt dabei das aktuelle Flagregister, das CS- und das IP-Register in den Stack. Der 8259 antwortet auf diesen Zyklus mit der Interruptnummer, in unserem Fall, Interrupt 9. Danach setzt der Controller das Bit 1 des ISR und zeigt damit an, daß Interrupt 1 in Arbeit ist. In der Zwischenzeit sucht der 8088 den CS:IP-Wert aus den Speicherstellen 24H und 27H und setzt die Programmausführung an dieser Stelle fort.

Der Prozessor führt nun die Tastaturunterbrechungsroutine aus. In Abbildung 8.4 sehen wir eine Skelettroutine zur Behandlung des Tastaturinterrupts. Halten wir dabei fest, daß es die erste Handlung der Interruptroutine ist, das AX-Register in den



```
The IBM Personal Computer MACRO Assembler 01-01-83           PAGE    1-1
Figure 8.4  Keyboard Control

 1                                  PAGE    ,132
 2                                  TITLE   Figure 8.4  Keyboard Control
 3
 4      0000                CODE    SEGMENT
 5                                  ASSUME  CS:CODE
 6
 7      0000                KEYBOARD_INTERRUPT      PROC    FAR
 8
 9      0000  50                    PUSH    AX              ; Save AX register
10      0001  E4 60                 IN      AL,60H          ; Get the keystroke value
11      0003  50                    PUSH    AX              ; Save the key value
12      0004  E4 61                 IN      AL,61H
13      0006  8A E0                 MOV     AH,AL           ; Save the current value
14      0008  0C 80                 OR      AL,80H
15      000A  E6 61                 OUT     61H,AL          ; Signal character received
16      000C  8A C4                 MOV     AL,AH
17      000E  E6 61                 OUT     61H,AL          ; Back to normal for keyboard
18      0010  58                    POP     AX              ; Recover keystroke
19
20                          ; . . . . .
21
22      0011  B0 20                 MOV     AL,20H
23      0013  E6 20                 OUT     20H,AL          ; Signal end of interrupt processing
24      0015  58                    POP     AX              ; Recover interrupted AX register
25      0016  CF                    IRET                    ; Return to interrupted program
26      0017                KEYBOARD_INTERRUPT      ENDP
27      0017                CODE    ENDS
28                                  END
```

Abbildung 8.4 Tastatursteuerung

Stack zu sichern. Dies wird deshalb getan, da wir im weiteren Verlauf das AX-Register modifizieren werden. Stellen wir den Originalwert dieses Registers vor der Rückkehr in das unterbrochene Programm nicht wieder her, so werden ganz sicher Programmfehler auftreten. Es ist schon schwer genug, korrekte Programme zu schreiben, ohne daß auch noch ein anderes Programm ganz zufällig diese Registerinhalte während des Ablaufs verändert. Benötigt unsere Interruptroutine nun mehr Register, so müssen wir auch diese sichern und später wieder rückspeichern.

Die Interruptroutine liest nun den Tastaturwert, der auch als Scan-Code bezeichnet wird, von Port 60H. Wir müssen nun den Interrupt löschen, indem wir dem Ein/Ausgabegerät mitteilen, daß es die Interruptanforderung rücksetzen soll. Unser Programm löscht nun den Tastaturinterrupt, indem wir Bit 7 an Port 61H umschalten. Dadurch wird nicht nur die Unterbrechungsanforderung gelöscht, sondern auch der Tastatur mitgeteilt, daß sie ein weiteres, also nächstes Zeichen an den 8088 senden kann. Die Unterbrechungsroutine kann nun den Wert der gedrückten Taste verarbeiten.

Hat die Unterbrechungsroutine ihre Arbeit abgeschlossen, sendet sie einen EOI-Befehl an den 8259. Dadurch wird das Bit 1 des ISR zurückgesetzt, so daß nun alle anderen Geräte mit niedrigerer Priorität, die vielleicht bereits warten, ihre Unterbrechungen anmelden können. Der EOI-Befehl besteht ganz einfach darin, daß wir den Wert 20H an den Port 20H senden. Außerdem stellt unsere Interruptroutine das Register AX wieder her und der Befehl IRET außerdem die Register IP, CS und das Flagregister.

Die gleiche Ereignisfolge trifft auch für jeden anderen Interrupt zu, der vom 8088 bearbeitet wird, wenn er ihn vom 8259 erhält. Der einzige Unterschied ist dabei die jeweilige Art der Unterbrechung. Dadurch ändern sich die entsprechenden Bits im IRR und ISR und der entsprechende Interruptvektor, der dem 8088 übergeben wird. Die Unterbrechungsroutine muß dabei immer den aktuellen Zustand des Prozessors zur Zeit der Unterbrechung aufbewahren. Dabei muß nach jeder Unterbrechungsbehandlungsroutine ein EOI-Befehl gegeben werden oder aber immer dann, wenn eine Unterbrechungsroutine festgestellt hat, daß das jeweilige Programm Unterbrechungen von niedrigerer Priorität annehmen kann. Es ist also Vorsicht geboten. Außerdem sollte unsere Unterbrechungsroutine keine zwei EOI-Befehle abgeben. Hat nämlich die gerade ablaufende Unterbrechungsroutine eine andere von niedrigerer Priorität unterbrochen, so entspricht der zweite EOI-Befehl dem EOI-Befehl, der eigentlich in der Routine mit der niedrigeren Priorität stehen müßte. Dies könnte zu unvorhersehbaren Ergebnissen führen, deshalb sollten wir es vermeiden.

Nachdem wir nun die Tastaturunterbrechung abgearbeitet, alle Bits gesetzt und alle Befehle korrekt ausgeführt haben, was haben wir damit erreicht? Die Tastatur sendet sogenannte Scan-Codes an den 8088. Diese Werte stellen die Position der Taste auf der Tastatur dar und haben nichts mit dem Zeichen zu tun, das wir auf der Taste sehen. So sendet beispielsweise die ESC-Taste den Scan-Code 1, die „1"-Taste den Scan-Code 2 usw. Die DEL-Taste liefert beispielsweise den Scan-Code 83. Jede Taste hat also ihren eigenen Scan-Code. Wir können die Daten an

Port 60H also nicht ganz einfach als ASCII-Zeichen interpretieren. Wir müssen erst diesen Scan-Code in den korrekten Zeichenwert übersetzen.

Die Tastatur sendet aber noch mehr als diese 83 Scan-Codes. Die ersten 83 Codes, von 1 bis 83, bezeichnen wir als Arbeitscodes. Die Tastatur sendet einen solchen Arbeitscode immer dann, wenn eine Taste gedrückt wird. Wird die Taste dagegen losgelassen, so sendet die Tastatur einen zweiten Code, den Ruhecode. Der Ruhecode wird von der Tastatur durch eine einfache Addition von 128 auf den Arbeitscode dargestellt. Auf diese Weise befinden sich die Ruhecodes im Bereich zwischen 129 und 211. Wir können die Ruhecodes außerdem einfach dadurch feststellen, daß Bit 7 gesetzt ist.

Da die Tastatur solchermaßen verschiedene Codes durch das Drücken und Wiederloslassen einer Taste sendet, können wir also jede Bewegung auf der Tastatur sofort erkennen. Wir erkennen, daß eine Taste gedrückt ist, da die Tastatur einen Arbeitscode sendet. Wir können also annehmen, daß die Taste so lange gedrückt bleibt, bis wir den Ruhecode empfangen, der anzeigt, daß die Taste wieder losgelassen wurde. Für die meisten Tasten ist diese Information allerdings nicht von großer Bedeutung. Bei den Shift-Tasten wird diese Information dagegen entscheidend. So werden beispielsweise Großbuchstaben nur dann erzeugt, wenn gleichzeitig zu einer gedrückten Taste auch die Shift-Taste gedrückt ist. Da wir die beiden Informationen, also Arbeits- und Ruhecode von jeder Taste der IBM-Tastatur erhalten, ist auch die Bearbeitung der Shift-Keys kein besonderes Problem. Wollen wir beispielsweise die Kodierung der Tastatur verändern, um etwa die Tastatur nach unseren eigenen Wünschen zu gestalten, so können wir jede Taste als Shift-Taste verwenden.

Außerdem verfügt jede Taste über eine Wiederholfunktion. Halten wir dabei eine Taste länger als 1/2 Sekunde gedrückt, so wird der von der Taste ausgehende Scan-Code jede Zehntelsekunde wiederholt. Dies ist von großer Hilfe für beinahe alle Tasten, speziell jedoch für die Tasten zur Steuerung des Cursors. Wir brauchen sie dabei nur einfach gedrückt zu halten, und der Cursor bewegt sich in die gewünschte Richtung. Handelt es sich dabei jedoch um eine Taste, die nur bei ihrem ersten Drücken eine bestimmte Funktion auslösen soll (und deshalb keinesfalls eine Wiederhol- oder Maschinengewehrfunktion auslösen darf), so müssen wir das Auf und Ab der Taste einzeln mitverfolgen. In diesem Fall dürfen wir nur den ersten Arbeitscode der Taste auswerten und müssen alle anderen Arbeitscodes ignorieren.

Tageszeit

Der Zeitgeberkanal 0 des 8253 dient auf dem IBM PC für einen ganz speziellen Zweck. Der Ausgang des Zeitgeberkanals ist nämlich mit der Interruptebene 0 des 8259 verbunden. Das bedeutet, daß immer dann eine Unterbrechung auftritt, wenn der Ausgangskanal 0 aktiv wird. Beim Einschalten des Systems wird dabei der Kanal 0 des Zeitgebers mit dem Anfangswert 0 geladen. Dies ergibt den größten (und nicht den kleinsten!) Wert, den unser Programm in diesem Zähler speichern kann. Mit einer Taktfrequenz von 1.19 MHz zählt dieser Zähler in etwas weniger als

55 Millisekunden zurück bis auf Null. Durch die Initialisierungsroutine wird dieser Zeitgeber außerdem so gesetzt, daß er ununterbrochen läuft. Dies bedeutet, daß der Interrupt 0 18.2 Mal pro Sekunde auftritt.

Wie wir im nächsten Kapitel sehen werden, wird dieser konstante Zeitgeberinterrupt vom ROM BIOS verwendet, um die laufende Zeit verfügbar zu halten. Nach jedem Auftreten des Zeitgeberinterrupts wird vom BIOS die Tageszeituhr um einen Tick weitergesetzt. Mit einem geeigneten Rechenvorgang können wir nun die Anzahl der Tick-Tack des Zeitgebers in Stunden, Minuten und Sekunden verwandeln.

Warum wurde nun genau der Wert 18.2 gewählt? Warum wurde der Zähler nicht so gesetzt, daß der Interrupt 20 Mal pro Sekunde auftritt oder zu einer anderen vernünftigen Zahl? Im nächsten Beispiel sehen wir, warum.

Mit dem Systemzeitgeber können wir auch noch andere Zeitfunktionen vorgeben als nur die Tageszeit. Die Tageszeit ist für uns sehr praktisch, wenn es sich um Zeitintervalle dreht, die in Sekunden oder Minuten gemessen werden. Handelt es sich jedoch um Steuerungssituationen in Ein/Ausgabegeräten, so werden zum Messen Zeitintervalle im Bereich zwischen 1 und 2 Millisekunden nötig. Normalerweise werden in einem Programm diese Zeitintervalle mittels einer Zeitschleife erzeugt. Ein Programm für eine Zeitschleife könnte etwa folgendermaßen aussehen:

```
        MOV     CX,LOOP_VALUE
HERE:   LOOP    HERE
```

Wir wählen dabei den konstanten Wert LOOP_VALUE so aus, daß die Schleife genau die richtige Anzahl von Malen durchlaufen wird. Dies ist eine sehr geeignete Methode, um eine Verzögerung von einer gewissen Zeit zu erreichen. In unserem angeführten Beispiel würde ein anfänglicher Schleifenwert LOOP_VALUE von 0FFFFH eine Verzögerung von etwa 250 Millisekunden bedeuten.

Nehmen wir nun einmal an, wir wollen ein externes Gerät überwachen und dazu feststellen, wie lange es dauert, bis ein bestimmtes Ereignis auftritt. Dazu könnten wir die folgende Variation der Zeitschleife verwenden:

```
        MOV      CX,0
HERE:            ; Test auf auftretendes Ereignis
        IN       AL,DX
        TEST     AL,MASK_BIT
        LOOPNE   HERE
DONE:            ; CX enthält die Anzahl der Schleifendurchläufe
```

Bei dieser Vorgehensweise zählen wir die Anzahl der Schleifendurchläufe, um festzustellen, wieviel Zeit bis zum Auftreten des Ereignisses verstrichen ist. Wir gehen dabei davon aus, daß das gewünschte Ereignis eintritt, bevor der Inhalt des CX-Registers zum zweiten Male 0 erreicht. Befindet sich dabei allerdings die Zeitauflösung, die wir für unsere Messung benötigen, im Bereich von Mikrosekunden, so können wir diese Methode nicht verwenden. Unsere Schleife benötigt nämlich für jeden Durchlauf zwischen 10 und 20 Mikrosekunden. Eine bessere Lösung

```
 The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.5  System Timer

1                                        PAGE    ,132
2                                        TITLE   Figure 8.5  System Timer
3      0000                      STACK   SEGMENT STACK
4      0000      40 [                    DW      64 DUP(?)
5                      ????
6                            ]
7
8      0080                      STACK   ENDS
9      0000                      CODE    SEGMENT
10                                       ASSUME  CS:CODE
11     0000                      TIMER   PROC    FAR
12     0000  1E                          PUSH    DS              ; Set return address
13     0001  B8 0000                     MOV     AX,0
14     0004  50                          PUSH    AX
15
16     0005  B0 B6                       MOV     AL,10110110B    ; Set timer 2
17     0007  E6 43                       OUT     43H,AL
18     0009  B8 0500                     MOV     AX,500H
19     000C  E6 42                       OUT     42H,AL          ; Timer 2 divisor will be 500H
20     000E  8A C4                       MOV     AL,AH
21     0010  E6 42                       OUT     42H,AL
22
23     0012  E8 001D R                   CALL    LOW_TO_HIGH     ; Get time of first low to high
24     0015  8B D8                       MOV     BX,AX           ; Save that value in BX
25     0017  E8 001D R                   CALL    LOW_TO_HIGH     ; Get time of second low to high
26     001A  2B D8                       SUB     BX,AX           ; Subtract to get cycle time
27     001C  CB                          RET
28     001D                      TIMER   ENDP
29                               ;--------------------------------
30                               ; This subroutine waits for the
31                               ;  first low-to-high transition
32                               ;  of the timer channel 2 output.
33                               ;  The timer 0 count at that time
34                               ;  is returned in AX.
35                               ;--------------------------------
36     001D                      LOW_TO_HIGH     PROC    NEAR
37     001D  E4 62                       IN      AL,62H          ; Test the timer channel 2 bit
38     001F  A8 20                       TEST    AL,20H
39     0021  75 FA                       JNZ     LOW_TO_HIGH     ; Loop until it is low
40     0023                      WAIT_HIGH:
41     0023  E4 62                       IN      AL,62H          ; Test the bit again
42     0025  A8 20                       TEST    AL,20H
43     0027  74 FA                       JZ      WAIT_HIGH       ; Loop until it goes high
44
45     0029  B0 00                       MOV     AL,0            ; Send command to timer control register
46     002B  E6 43                       OUT     43H,AL          ;  that "freezes" the count in timer 0
47     002D  90                          NOP
48     002E  90                          NOP                     ; Delay necessary for 8253
49     002F  E4 40                       IN      AL,40H
50     0031  8A E0                       MOV     AH,AL           ; Read the low byte of the count
51     0033  90                          NOP
52     0034  E4 40                       IN      AL,40H          ; Read the high byte of the count
53     0036  86 E0                       XCHG    AH,AL           ; Move into correct positions
54     0038  C3                          RET                     ; Return current count in AX
55     0039                      LOW_TO_HIGH     ENDP
56     0039                      CODE    ENDS
57                                       END
```

Abbildung 8.5 Systemzeitgeber

erreichen wir über den Systemzeitgeber. Da nämlich der Wert dieses Zeitgebers alle 840 Nanosekunden wechselt, können wir Zeitspannen bis zu einer Mikrosekunde messen.

In Abbildung 8.5 sehen wir ein Beispielprogramm, bei dem zur Zeitmessung der Systemzeitgeber herangezogen wird. Dazu wird der Zeitgeberausgang 2 als das zu messende Ereignis verwendet. Im ersten Teil des Programms wird dazu der Zeitgeberkanal 2 auf einen bekannten Wert gesetzt. Wir wählten dazu willkürlich den Wert 500H. Halten wir fest, daß dieser Programmabschnitt identisch ist mit der Tonerzeugung über Zeitgeberkanal 2.

Unser Programm ruft nun ein Unterprogramm, LOW_TO_HIGH, auf, das den aktuellen Wert des Zeitgebers zurückgibt, wenn ein Übergang vom Low- auf den High-Zustand am Ausgang des Zeitgeberkanals 2 auftritt. Unser Programm achtet dabei nur auf den Signalwechsel. Würden wir nämlich nur auf das High-Signal sehen, könnten wir nicht erkennen, ob unser Signal gerade erst den High-Wert erreicht hat oder ob es gerade dabei ist, wieder den Low-Zustand einzunehmen. Zu diesem

Zweck sendet unser Unterprogramm den Wert 0 an das Zeitgeber-Kontrollregister (Port 43H), um den gerade aktuellen Wert des Zeitgebers 0 „einzufrieren". Dies erlaubt es uns, den aktuellen Wert des Zeitgebers zu lesen, während der Zeitgeber selbst weiterzählt. Könnten wir nämlich den Zeitgeber nicht vorübergehend einfrieren, so könnten wir auch niemals den 16-Bit Wert des Zeitgebers korrekt lesen.

Halten wir außerdem fest, daß das Unterprogramm in Abbildung 8.5 einige NOP-Befehle enthält. Diese Befehle sind nur aus Zeitgründen eingefügt. Lesen wir nämlich die Spezifikationen des 8253 sorgfältig durch, so werden wir erkennen, daß mindestens eine Mikrosekunde Zwischenraum zwischen den Ein- und Ausgabebefehlen für diesen Chip vorhanden sein muß. Die NOP-Befehle dienen nun dazu, sicherzustellen, daß wir diese Zeitbedingungen für den Baustein erfüllen.

Nach Rückkehr aus unserem Unterprogramm sichern wir im BX-Register den Wert des Zeitgebers beim ersten Übergang von Low auf High. Dann ruft unser Programm wiederum das Unterprogramm auf, um den nächsten Low-High-Übergang des Zeitgeberkanals 2 festzustellen. Wir subtrahieren dann die beiden Werte voneinander, um die Zykluszeit des Zeitgeberkanals 2 festzustellen.

Wir haben bereits erwähnt, daß es sinnvoll ist, das Zeitgeberregister 0 mit dem Anfangswert 0 vorzubesetzen. Unser Programm ist nun der Beweis hierfür, denn wir subtrahieren dabei zwei Zeitgeberwerte, ohne Rücksicht darauf, welcher von den beiden größer oder kleiner ist. Da der Zeitgeber 0 nämlich asynchron zur Ausführung unseres Programms läuft, ist keine Garantie dafür gegeben, daß die erste gelesene Zahl immer größer als die zweite ist. Nehmen wir beispielsweise einmal an, daß der erste Low-High-Übergang auftritt, wenn der Timer 0 auf dem Wert 100H steht. Nach 500H Zählschritten steht dann im Zeitgeber 0 der Wert 0FC00H. Der Timer 0 kippt nämlich nach einem Nulldurchgang automatisch auf 0FFFFH um. Auf diese Weise ist der zweite Wert numerisch größer als der erste. Da das Zeitgeberregister jedoch nach jedem Null-Durchgang wieder mit 0FFFFH zu zählen beginnt, können wir immer zwei Zahlen voneinander subtrahieren. Manchmal wird es dabei einen Übertrag ergeben, manchmal auch keinen. Die Differenz der beiden Zahlen wird jedoch immer so groß sein wie die Anzahl der verstrichenen Zeitgeberimpulse.

Um den Wahrheitsbeweis anzutreten, erinnern wir uns an den Fall, in dem der Zähler mit dem Wert 8000H initialisiert wurde. Würde das erste Umschalten nun bei einem Wert von 6000H passieren und das zweite bei einem Wert von 5B00H, so wäre die Differenz 500H. Würde dagegen der erste Umschaltvorgang bei einem Zählerstand von 100H auftreten, so würde der zweite Umschaltvorgang bei einem Zählerstand von 7C00H passieren. In diesem Falle wäre die Differenz 8500H. Um diese Situation nun sicher zu beherrschen, wäre es in unserem Programm nötig, einen solchen Nulldurchlauf des Zeitgebers auf irgendeine Weise herauszufinden.

Lassen wir nun dieses Programm laufen, so werden wir feststellen, daß sich im BX-Register ein Wert befindet, der in etwa 0A00H entspricht. Dies sind in keinem Falle die 500H, die wir eigentlich dort erwarteten. Dies ist darauf zurückzuführen, daß sich der Zeitgeber in einem Modus befindet, bei dem er den Zählerwert für jeden Taktimpuls um jeweils 2 erniedrigt. Um die Arbeitsweise des 8253 wirklich zu verstehen, müssen wir also sein Programmierhandbuch konsultieren.

Format des Steuerworts

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

Steuerung

SC – Zähler auswählen:

SC1	SC0	
0	0	Zähler 0
0	1	Zähler 1
1	0	Zähler 2
1	1	unzulässig

RL – Lesen/Laden:

RL1	RL0	
0	0	Zähler laden (Vorgehen wie bei Lesen/Laden)
1	0	Lesen/Laden höherwertiges Byte
0	1	Lesen/Laden niederwertiges Byte
1	1	Lesen/Laden niederwertiges, dann höherwertiges Byte

M – Modus:

M2	M1	M0	
0	0	0	Modus 0
0	0	1	Modus 1
X	1	0	Modus 2
X	1	1	Modus 3
1	0	0	Modus 4
1	0	1	Modus 5

BCD:

0	Binärzähler 16 Bits
1	BCD-Zähler (4 Dekaden)

Abbildung 8.6 Programmierung von Zeitgeber/Zählern
(mit freundlicher Genehmigung von Intel; Copyright Intel 1981)

In Abbildung 8.6 ist das Kontrollwort für den 8253 zusammengefaßt. Wir geben dieses Kontrollwort auf Port 43H aus, um einen der gewünschten Kanäle für eine bestimmte Operation vorzubereiten. Wir haben bereits anhand verschiedener Bei-

spiele gesehen, wie Werte auf den Port 43H ausgegeben werden. So sandten wir beispielsweise den Wert 0 auf diesen Port, um den Zähler einzufrieren oder den Wert 0B6H, um einen Ton zu erzeugen. Sehen wir uns nun einmal an, woher diese Werte stammen.

Die zwei höchstwertigen Bits des Kontrollworts wählen den Timerkanal aus. Die nächsten beiden Bits bestimmen die gewünschte Operation. Als wir also eine 0 ausgaben, wählten wir damit den Zeitgeberkanal 0 und besetzten außerdem den Zähler vor. Die nächsten 3 Bits dienen zur Auswahl der Arbeitsweise des gewünschten Timers. Diese Bits benötigen wir allerdings nicht, wenn wir nur den Zähler vorbesetzen, wir benötigen sie jedoch, wenn der Zeitgeber initialisiert werden soll. Mit dem letzten Bit legen wir schließlich noch fest, ob der Zähler als 16-Bit-Binärzähler oder als vierziffriger BCD-Zähler läuft.

Der Steuerwert 0B6H, den wir für die Tonerzeugung verwendeten, setzt sich also wie folgt zusammen:

0B6H = 10110110B = 10 11 011 0

Als erstes wählen wir den Zeitgeber 2 aus. Der Zähler dieses Zeitgebers arbeitet mit einem 16-Bit Wert, wobei wir das niederwertige Byte dieses Werts zuerst in den Zähler laden müssen. Das letzte Bit gibt schließlich noch an, daß wir binär zählen wollen. Außerdem können wir sehen, daß der Verarbeitungsmodus 3 ausgewählt wurde.

Der 8253 kann auf sechs verschiedene Arten betrieben werden. Für unsere Zwecke sind dabei auf dem IBM-System nur zwei verwendbar. Die Verarbeitungsart 3 ist der Standardweg, auf dem alle drei Kanäle des Zeitgebers verwendet werden. Der Zeitgeber arbeitet dann als Rechteckgenerator. Die Hälfte einer Zyklusdauer lang ist der Ausgang des Zeitgebers dabei auf Low und die andere Hälfte auf High. Dies wird dadurch erreicht, daß der Zählschritt 2 und nicht 1 beträgt, wie beispielsweise beim Zählen von einem festgelegten Wert. Während des ersten Zählschritts ist der Ausgang dabei Low und während des zweiten High. Da der Zähler insgesamt in Zweierschritten zählt, ist der Ausgang also während der ersten Hälfte der Zeit Low und während der zweiten Hälfte High. Da der Zeitgeber 0 normalerweise in diesem Modus 3 läuft, erzeugte das Beispiel in Abbildung 8.5 auch den Endwert 0A00H anstelle des erwarteten Werts 500H. Bei jedem Zählschritt wurde nämlich 2 vom Zähler abgezogen.

Da der Zähler solchermaßen in Zweierschritten zählt, erfolgt alle 27 Mikrosekunden ein Nulldurchlauf. Wollen wir nun Ereignisse messen, die länger als 27 Mikrosekunden dauern, so werden wir uns eine andere Methode überlegen müssen.

Diese zweite Methode, die auch im IBM PC verwendet wird, ist der Modus 0. Dieser Modus wird auch als Zeitgeberinterrupt bezeichnet. Der Zeitgeber läuft dabei nicht dauernd. Haben wir nämlich als erstes den Modus ausgewählt, beginnt der Zeitgeber so lange nicht mit der Arbeit, also mit dem Zählen (in Einerschritten), bis nicht der vollständige Ausgangswert in das Zeitgeberregister geladen ist. Der Zähler zählt dann im Rythmus der Taktfrequenz abwärts, bis das Zählerregister schließlich den Wert 0 erreicht. In diesem Augenblick wird der Ausgang des Zeitgebers High.

Da der Ausgang des Zeitgebers 0 aber mit dem Interrupt 0 des 8259 verbunden ist, tritt eine Unterbrechung auf.

Dieser Zeitgeberinterrupt ist von Nutzen, wenn unser Programm nach Ablauf einer bestimmten Zeit eine Unterbrechung auslösen soll. Da der Zähler dabei auf 16 Bits begrenzt ist, beträgt das maximale Zeitintervall 55 Millisekunden. Ist dieses Intervall für unsere Zwecke zu kurz, benötigen wir noch eine dritte Methode der Zeitmessung.

Wollen wir einen Zeitraum in Sekunden messen, so sollten wir den Timer in seiner normalen Arbeitsweise belassen. Das ROM BIOS gibt uns nämlich die Möglichkeit, alle 55 Millisekunden die Steuerung zu übernehmen. Und bei jedem „Ticken" des Systemzeitgebers können wir entscheiden, ob die von uns gewünschte Zeitspanne bereits verstrichen ist.

Befindet sich der gewünschte Zeitraum irgendwo im Bereich zwischen 55 Millisekunden und 5 Sekunden, so könnten wir noch einen zusätzlichen Weg zur Zeitmessung gehen, ohne das ROM BIOS zu verwenden. Nehmen wir dazu einmal eine Zeitspanne von 150 Millisekunden. Im Zeitgeberinterrupt-Modus können wir den Zeitgeber so steuern, daß alle 50 Millisekunden ein Interrupt auftritt (dies benötigt einen Anfangswert von 59500). Wir programmieren dann die Interruptroutine so, daß der Timer bei den ersten beiden Zeitgeberunterbrechungen wieder um 50 Millisekunden rückgesetzt wird. Nach dem dritten Zeitgeberinterrupt sind 150 Millisekunden verstrichen, und wir können die gewünschte Funktion ausführen.

Bei allen Verwendungen des Timers sollten wir jedoch Vorsicht walten lassen. Wie wir bereits erwähnten, führt Timer 1 eine wichtige Hardwarefunktion aus. Sollten Sie versuchen, den Wert in Timer 1 zu verändern, so wird Ihr Programm auf der Stelle sterben. Die Verwendung des Zeitgebers 2 ist dagegen wesentlich vernünftiger. Der Timer 2 ist nämlich nur mit dem Lautsprecher und mit dem Kassettenausgang verbunden. Das bedeutet allerdings auch, daß wir nicht versuchen sollten, den Timer 2 für Zeitgeberaufgaben zu verwenden, wenn wir gleichzeitig auf dem Lautsprecher Musik erzeugen wollen. Schließlich verwendet noch das ROM BIOS den Timer 0 für verschiedene Systemfunktionen. Wie wir später bei der Diskussion des ROM BIOS noch sehen werden, steuert der Tageszeitinterrupt nicht nur die Uhr, sondern auch den Motor des Diskettenlaufwerks. Bevor wir also dem Zeitgeber 0 eine neue Aufgabe zuweisen, müssen wir zuerst die Funktionen verstehen, die er ausführt, bevor wir sie verändern.

Systemeigenschaften

In diesem Abschnitt besprechen wir die Programmieraspekte einiger Systemeigenschaften. Nicht jedes System verfügt dabei notwendigerweise über die Eigenschaften, die wir im weiteren beschreiben werden. Wir werden jedoch Eigenschaften beschreiben, die Sie auf vielen Systemen antreffen werden. Unsere Beschreibung umfaßt im folgenden die beiden Bildschirmadapter, den Diskettenadapter, den Druckeradapter, den Asynchronadapter und den Spieladapter.

Bildschirmadapter

Sie können den IBM PC mit zwei verschiedenen Bildschirmadaptern kaufen. Einer wird als Monochrome Display and Printer Adapter bezeichnet (wir werden ihn als Monochromadapter oder auch ganz einach als Schwarz/Weiß-Karte bezeichnen). Dieser Bildschirmadapter ist besonders geeignet für Rechneranwendungen im Geschäftsbereich. Die Schwarz/Weiß-Karte dient zur Steuerung des IBM Monochrom-Bildschirms. Der Monochromadapter kann nur in einem einzigen Modus arbeiten und dient nur zur Wiedergabe einfarbigen Textes auf dem Bildschirm. Der Adapter verfügt zusätzlich über die nötige Hardware, um einen Drucker zu betreiben. Diese Kombination macht die Adapterkarte für geschäftliche Anwendungen besonders attraktiv.

Die zweite Bildschirmadapterkarte wird bezeichnet als Color/Graphics Monitor Adapter (wir bezeichnen sie ganz einfach als Farbkarte). Die Farbkarte wird verwendet für Bildschirme, die mit normalen Fernsehfrequenzen arbeiten. Im Gegensatz zu den einfarbigen Bildern des Monochromadapters ist die Farbkarte auch geeignet, farbige Bilder zu erzeugen. Dazu verfügt sie über verschiedene Arbeitsmodi einschließlich solcher, in denen wir jeden einzelnen Bildschirmpunkt ansprechen können.

Schwarz/Weiß-Bildschirm- und Druckeradapter

Als erstes werden wir die Schwarz/Weiß-Karte besprechen. Sie ist die einfachere der beiden Adapterkarten. Die Besprechung des Druckersteuerungsteils dieser Karte stellen wir zurück, bis wir den eigentlichen Druckeradapter besprechen. Die Schaltkreise zur Druckersteuerung auf der Monochromkarte sind nämlich identisch zum einfachen Druckeradapter, so daß also auch ihre Programmierung gleich abläuft.

Die Schwarz/Weiß-Karte arbeitet nur in einem einzigen Modus. Die Bildschirmsteuerung erzeugt dabei 25 Buchstaben-Zeilen, die jeweils 80 Zeichen enthalten und gemeinhin als 80x25-Schirm bezeichnet werden. Die Darstellung von Zeichen auf dem Bildschirm geschieht durch einfaches Abspeichern von ASCII-Werten im Bildschirmpuffer. Dieser Bildschirmpuffer ist ein besonderer Speicherbereich, der bei der Adresse 0B0000H beginnt. Dieser Speicher befindet sich auf der Adapterkarte und ist demzufolge nicht Teil des Systemspeichers. Legen wir nun den ASCII-Wert für einen Buchstaben im Bildschirmspeicher ab, so erscheint dieses Zeichen an der korrespondierenden Stelle auf dem Bildschirm. Die Umsetzung des Zeichens aus der ASCII-Darstellung in die einzelnen Bildschirmpunkte wird dabei von der Hardware ausgeführt.

Jedes Zeichen auf dem Bildschirm verfügt zusätzlich über ein Attribut. Diese Attribute bestimmen die Art und Weise, in der die Adapterkarte die einzelnen Zeichen auf dem Bildschirm darstellt. In Abbildung 8.7 sehen wir die einzelnen Zeichenattribute und die Werte, die für sie nötig sind. Wir müssen diese Werte kennen, denn auch sie müssen wir selbst an die entsprechenden Stellen im Bildschirmpuffer

ablegen. Jede Zeichenposition innerhalb des Bildschirmpuffers besteht dabei aus zwei Bytes. Das geradzahlige Byte eines solchen Paares enthält dabei den Zeichenwert und das andere Byte den Attributwert. Über Abbildung 8.7 können wir ermitteln, welchen Attributwert wir beispielsweise für einen Buchstaben ablegen müssen. Normalerweise verwenden wir dabei weiße (in Wirklichkeit grüne) Zeichen auf schwarzem Hintergrund. Der Attributwert hierfür ist 07H. Wollen wir die Zeichen invertiert darstellen, so ändern wir einfach den Attributwert in 70H. Ein Attributwert von 00H würde unsere Zeichen unsichtbar machen. Selbst wenn wir in diesem Fall einen ASCII-Code für ein darstellbares Zeichen im Bildspeicher ablegen würden, würde das Attribut verhindern, daß dieses Zeichen auf dem Bildschirm erscheint.

Der Bildpuffer für die Schwarz/Weiß-Karte beträgt 4K. Dies genügt zur Speicherung sowohl von Zeichenbyte als auch Attributbyte für jede der 2000 möglichen Zeichenpositionen auf dem Bildschirm. Das erste Zeichen des Bildschirmpuffers erscheint dabei in der oberen linken Ecke des Bildschirm. Die nächsten beiden Bytes entsprechen dem nächsten, nach rechts anschließenden Zeichen usw. Das erste Zeichen in der zweiten Zeile des Bildschirms würde sich also in den Bytes 160 und 161 des Bildschirmpuffers befinden. Solchermaßen können wir die Adresse jeder Zeichenposition des Bildschirms bestimmen. Dazu definieren wir als erstes die Zeichenposition in der linken oberen Ecke des Bildschirms als Zeile 0 und Spalte 0. Das rechte untere Eck des Bildschirms wäre dann Zeile 24 und Spalte 79. Die Formel für eine beliebige Spalte und Zeile ist dann:

Adresse = 2 × (Zeile x 80 + Spalte) + 0B0000H

Die Multiplikation mit 2 sorgt dabei für die nötige Ausrichtung auf jeweils 2 Bytes pro Zeichenposition. Die Addition mit 0B0000H gibt die Startadresse des Bildschirmpuffers wieder. Normalerweise setzen wir dabei in unserem Programm das DS- oder ES-Register auf den Wert 0B000H und arbeiten dann im weiteren im Programm nur noch mit dem Offset innerhalb des Bildschirmpuffers.

Wert	Attribut
00H	keine Ausgabe
01H	unterstrichene Zeichen
07H	weißes Zeichen auf schwarzem Grund
0FH	intensiv weißes Zeichen auf schwarzem Grund
70H	schwarzes Zeichen auf weißem Grund
80H	bei Addition dieses Wertes auf einen der oben angeführten blinkt das jeweilige Zeichen

Abbildung 8.7 Zeichenattribute für die Schwarz/Weiß-Karte

Als Beispiel für die Verwendung der Schwarz/Weiß-Karte bewegen wir im Programm in Abbildung 8.8 den Inhalt einer Bildschirmzeile um eine Position nach rechts. Während dabei die Spalte rechts außen verschwindet, wird die Spalte links außen auf Blank gesetzt. Sollten Sie dieses Programm versuchen wollen und nur über eine Farbkarte in Ihrem System verfügen, dann wird dieses Programm erst dann laufen, wenn Sie das DISPLAY-Segment auf die Adresse 0B800H setzen. Die beiden Adapterkarten verarbeiten nämlich Textdaten völlig gleich, unterscheiden sich aber in den Speicher- und I/O-Adressen.

```
 The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.8 Horizontal Scroll -- Right

1                                               PAGE    ,132
2                                               TITLE   Figure 8.8 Horizontal Scroll -- Right
3
4       0000                            STACK   SEGMENT STACK
5       0000      40 [                          DW      64 DUP(?)
6                       ????
7                              ]
8
9       0080                            STACK   ENDS
10
11      0000                            DISPLAY SEGMENT AT 0B000H
12      0000                            DISPLAY_START   LABEL   WORD
13      0FA0                                    ORG     4000
14      0FA0                            DISPLAY_END     LABEL   WORD
15      0FA0                            DISPLAY ENDS
16
17      0000                            CODE    SEGMENT
18                                              ASSUME  CS:CODE
19      0000                            SIDE_SCROLL     PROC    FAR
20      0000  1E                                PUSH    DS
21      0001  B8 0000                           MOV     AX,0
22      0004  50                                PUSH    AX
23      0005  B8  ---- R                        MOV     AX,DISPLAY
24      0008  8E D8                             MOV     DS,AX
25      000A  8E C0                             MOV     ES,AX
26                                              ASSUME  DS:DISPLAY,ES:DISPLAY
27
28                                      ;----- Fill column 79 with blanks
29
30      000C  B9 0019                           MOV     CX,25
31      000F  8D 3E 004E R                      LEA     DI,DISPLAY_START+78
32      0013  B8 0720                           MOV     AX,0720H
33      0016                            BLANK:
34      0016  89 05                             MOV     [DI],AX
35      0018  81 C7 00A0                        ADD     DI,160
36      001C  E2 F8                             LOOP    BLANK
37
38                                      ;----- Right shift
39
40      001E  B9 07D0                           MOV     CX,2000
41      0021  8D 36 0F9E R                      LEA     SI,DISPLAY_END-2
42      0025  8D 3E 0FA0 R                      LEA     DI,DISPLAY_END
43      0029  FD                                STD
44      002A  F3/ A5                            REP     MOVSW
45
46      002C  A3 0000 R                         MOV     DISPLAY_START,AX
47
48      002F  CB                                RET
49      0030                            SIDE_SCROLL     ENDP
50      0030                            CODE    ENDS
51                                              END
```

Abbildung 8.8 Waagrechtes Verschieben des Bildinhalts nach rechts

Das Programm im Abbildung 8.8 bewirkt dabei das waagrechte Verschieben der Daten auf eine sehr einfache Weise. Als erstes wird die rechts außen befindliche Spalte einer jeden Zeile auf Blank gesetzt, indem wir 25 Blanks im Abstand von jeweils 160 Bytes im Speicher ablegen. Darauf wird der gesamte Bildschirmpuffer im Speicher um ein Byte nach oben geschoben. Da im Bildschirmpuffer die Zeichen direkt zeilenweise hintereinander angeordnet sind, wird die Spalte 79 der Zeile 0 nun zur Spalte 0 der Zeile 1. Abschließend ersetzen wir noch das erste Zeichen im Bildschirmpuffer durch ein Blank.

Die Schwarz/Weiß-Karte verfügt noch über einige zusätzliche Ein/Ausgabeports, die wir zur Steuerung des Bildschirms verwenden können. Allerdings werden wir diese Ports nur ganz kurz besprechen. Auf der Schwarz/Weiß-Karte sind diese Ein/Ausgabeports nämlich hauptsächlich aus Gründen der Hardwarekompatibilität vorhanden. Außerdem entspricht das Programmieren der Ein/Ausgabe für die Schwarz/Weiß-Karte genau dem der Farbkarte. Da es aber bei der Farbkarte wesentlich mehr Möglichkeiten gibt, die Ein/Ausgabeports zu verwenden, werden wir sie dort genauer besprechen.

Die Schwarz/Weiß-Karte erzeugt zusätzlich die horizontalen und vertikalen Zeitsignale, die zur Bildschirmsteuerung mit dem Motorola 6845 CRT notwendig sind. Dieser Chip verfügt über zwei Ein/Ausgabeadressen, die Ein/Ausgabeports 3B4H und 3B5H. Außerdem verfügt unsere Adapterkarte über den Steuerport 3B8H und den Statusport 3BAH. Wird die Karte allerdings einmal initialisiert, gibt es nur noch wenige Gründe, die Werte in diesen Ports zu verändern. Es gibt noch einige andere Arten der Verarbeitung mit der Schwarz/Weiß-Karte, doch wir werden uns im weiteren auf die Farbkarte konzentrieren. Genaueres, besonders über die Ein/Ausgabeports der Schwarz/Weiß-Karte, können Sie im Technical Reference Manual finden.

Farb/Graphik-Monitoradapter

Der Farb- bzw. Graphik-Monitoradapter ist die andere Steuerkarte, die für den IBM PC lieferbar ist. Die Karte wurde von IBM außerdem so entwickelt, daß sie zur Steuerung von fernsehähnlichen Geräten verwendet werden kann. Während die Schwarz/Weiß-Karte nur mit dem Original-IBM Monochrombildschirm arbeitet, können wir an die Farbkarte jeden beliebigen Farb- oder Schwarz/Weiß-Monitor anschließen, der über Standardfernsehsignale verfügt. Außerdem besteht die Möglichkeit, mittels eines zusätzlichen Rundfunkmodulators die Farbkarte mit Ihrem häuslichen Fernsehgerät zu verbinden. Die Farbkarte kann außerdem mit fast allen gängigen Monitoren verbunden werden, da sie auf der Basis von Fernsehfrequenzen arbeitet.

Die Farbkarte verfügt über viele verschiedene Operationsmöglichkeiten. Von IBM selbst werden nur einige dieser Betriebsarten unterstützt. Die übrigen müssen Sie, wenn Sie sie verwenden wollen, selbst programmieren. Die Farbkarte verfügt also, im Gegensatz zur Schwarz/Weiß-Karte, über ein großes zusätzliches Maß an Flexibilität.

Zwei verschiedene Arten der Textdarstellung sind in Verbindung mit der Farbkarte möglich. Beim ersten dieser beiden Textmodi findet ein Bildschirm mit 25 Zeilen zu je 80 Zeichen Verwendung, entsprechend der Schwarz/Weiß-Karte. Eine zweite mögliche Art der Textdarstellung reduziert die Anzahl der Zeichen pro Zeile auf 40. Diese Arbeitsweise mit geringerer Auflösung ist notwendig, wenn Sie die Farbkarte in Verbindung mit einem niedrig auflösenden Monitor oder mit Ihrem häuslichen Fernsehgerät verwenden wollen. Sie werden nämlich feststellen, daß bei einem Bild mit 80 Zeichen auf einem Fernsehschirm die Zeichen manchmal unleserlich werden. Um die vollen Fähigkeiten der Farbkarte auszuschöpfen, müssen Sie also einen Qualitätsmonitor verwenden. Ein hochauflösender Monitor erlaubt es Ihnen dabei, die ganzen 80 Zeichen langen Zeilen ohne irgendwelche Verzerrungen abzubilden.

Die Farbkarte verfügt außerdem über zwei Arten der Graphikdarstellung. In beiden Modi können wir dabei jeden einzelnen Punkt auf dem Schirm ansprechen. Im Textmodus ist jedes einzelne Zeichen ansprechbar und im Graphikmodus jeder einzelne Punkt. Im Graphikmodus mit mittlerer Auflösung stehen uns dabei 200 Zeilen mit jeweils 320 Punkten zur Verfügung. Jeder dieser Punkte kann eine von vier mög-

lichen Farben annehmen. Das heißt, wir benötigen für jeden darstellbaren Punkt jeweils zwei Bits zur Steuerung der vier möglichen Farben. Im Graphikmodus mit hoher Auflösung stehen uns 200 Zeilen mit je 640 Punkten zur Verfügung. In diesem Modus können Sie allerdings nur noch zwei Farben, nämlich Schwarz und Weiß, verwenden.

Textmodus

Die beiden Textmodi der Farbkarte verfügen über einen einfachen Mechanismus, um Zeichen auf dem Bildschirm darzustellen. In den beiden Textmodi arbeitet die Farbkarte nämlich ebenso wie die Schwarz/Weiß-Karte. Wir können also für jedes einzelne Zeichen die gewünschte Farbe angeben. Analog zur Schwarz/Weiß-Karte verfügt auch die Farbkarte über jeweils ein Attributbyte für jedes Zeichen. Dieses Attributbyte bestimmt dann die Vordergrund- und Hintergrundfarben für das darzustellende Zeichen.

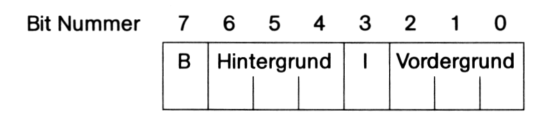


Abbildung 8.9 Attributbyte der Farbkarte

In Abbildung 8.9 sehen wir die Aufschlüsselung des Farbattributs im Textmodus. Die mit „Hintergrund" bezeichneten drei Bits legen dabei die Hintergrundfarbe fest, eine von acht möglichen Farben. Analog dazu geben die drei Bits, die mit „Vordergrund" bezeichnet sind, die Vordergrundfarbe an. Das mit I bezeichnete Bit bezieht sich nur auf den Vordergrund. Das Setzen dieses Bits bewirkt, daß die Vordergrundfarbe mit hoher Intensität dargestellt wird. Das ermöglicht es uns, bei den Vordergrundfarben unter 16 möglichen auszuwählen. Das Attributbyte bezieht sich dabei jeweils auf ein einziges Zeichen, so daß wir für jede mögliche Zeichenposition eine beliebige Kombination aus Vordergrund und Hintergrund verwenden können. Das höchstwertige Bit, mit B bezeichnet, wird normalerweise dazu verwendet, um die Vordergrundfarben blinkend darzustellen. Das Setzen dieses Bits auf 1 bewirkt, daß die Farbe des dargestellten Zeichens zwischen Vordergrund und Hintergrund wechselt, und zwar etwa viermal pro Sekunde. Da aber ein mit der Hintergrundfarbe dargestelltes Zeichen unsichtbar wird, ist das Endergebnis ein Blinken des dargestellten Zeichens. Wir können dieses Blink-Bit aber auch als viertes Bit zur Darstellung der Hintergrundfarbe verwenden, was uns dann 16 Hintergrund- und 16 Vordergrundfarben ermöglicht. Diese Umschaltung erfolgt durch ein Bit im Farbwahlregister. Außerdem ist zu bemerken, daß das Setzen von Hintergrund und Vordergrund auf den gleichen Farbwert bedeutet, daß das Zeichen für uns unsichtbar wird. Das Zeichen wird zwar ausgegeben, aber es verhält sich damit wie mit der Suche eines Eisbergs im Schneesturm — alles hat dieselbe Farbe. Im Abbildung 8.10 sehen wir die 16 möglichen Farben im Textmodus.

I	R	G	B	Farbe
0	0	0	0	schwarz
0	0	0	1	blau
0	0	1	0	grün
0	0	1	1	cyan
0	1	0	0	rot
0	1	0	1	magenta
0	1	1	0	braun
0	1	1	1	hellgrau
1	0	0	0	dunkelgrau
1	0	0	1	hellblau
1	0	1	0	hellgrün
1	0	1	1	hellcyan
1	1	0	0	hellrot
1	1	0	1	hellmagenta
1	1	1	0	gelb
1	1	1	1	weiß

Abbildung 8.10 Farben
(mit freundlicher Genehmigung der IBM, Copyright IBM 1981)

Vergleichen wir nun ein Attributbyte der Farbkarte mit dem Schwarz/Weiß-Attribut in Abbildung 8.7, so werden wir sehen, daß diese gleich sind. Natürlich können wir beim Schwarz/Weiß-Bildschirm keine Farben angeben, doch alles andere paßt genau. Da es auf der Farbkarte keine Unterstreichung gibt, müssen wir jedoch das Attribut auf blauen Vordergrund mit schwarzem Hintergrund setzen, um auf der Schwarz-Weiß-Karte eine Unterstreichung zu erzeugen.

Auch die Belegung des Attributbytes ist ein zusätzlicher Versuch, die beiden Bildschirmsteuerkarten so ähnlich wie möglich zu halten. Jedes Zeichen befindet sich wiederum an den geradzahligen Adressen im Bildspeicher und das Attribut jeweils an der darauffolgenden. Der Bildpuffer für den Farbbildschirm befindet sich auf der Adapterkarte, hat aber eine andere Speicheradresse als bei der Schwarz/Weiß-Karte. Der Bildschirmpuffer für den Schwarz/Weiß-Bildschirm beginnt dabei an der Speicheradresse 0B0000H, während der Bildschirmpuffer für den Farbbildschirm an der Speicheradresse 0B8000H beginnt. Um zu zeigen, wie ähnlich die beiden Steuerkarten sind, modifizieren wir einfach das Programm zur Ausgabe auf dem Schwarz/Weiß-Bildschirm in Abbildung 8.8, indem wir die Segmentadresse bei AT auf 0B800H setzen und sehen, daß unser Programm sofort korrekt auf der Farbkarte läuft. Ein Programm kann also mit einem Minimum an Änderungsaufwand von einer Karte auf die andere übertragen werden.

Auch die Farbkarte verwendet den CRT-Controller 6845 zur Steuerung des Adapters. Die beiden Ein/Ausgabeports des Controllers befinden sich dabei an den Ein/Ausgabeadressen 3D4H und 3D5H. Der 6845 verfügt jedoch in Wahrheit über 18 interne Register. Über die beiden Ein/Ausgabeports können wir aber alle Register mittels indirekter Adressierung erreichen. Um ein Register des 6845 zu adressieren, laden wir als erstes den Registerindex in den Ausgabeport 03D4H. Danach lesen oder schreiben wir den Port 3D5H, um direkt auf das entsprechend ausgewählte Register zuzugreifen.

Register #	Register File	Program Unit	Read	Write
R0	Horizontal Total	Char.	No	Yes
R1	Horizontal Displayed	Char.	No	Yes
R2	H. Sync Position	Char.	No	Yes
R3	H. Sync Width	Char.	No	Yes
R4	Vertical Total	Char. Row	No	Yes
R5	V. Total Adjust	Scan Line	No	Yes
R6	Vertical Displayed	Char. Row	No	Yes
R7	V. Sync Position	Char. Row	No	Yes
R8	Interlace Mode	–	No	Yes
R9	Max Scan Line Address	Scan Line	No	Yes
R10	Cursor Start	Scan Line	No	Yes
R11	Cursor End	Scan Line	No	Yes
R12	Start Address (H)	–	No	Yes
R13	Start Address (L)	–	No	Yes
R14	Cursor (H)	–	Yes	Yes
R15	Cursor (L)	–	Yes	Yes
R16	Light Pen (H)	–	Yes	No
R17	Light Pen (L)	–	Yes	No

Abbildung 8.11 Register des 6845
(mit freundlicher Genehmigung von Motorola, Inc.)

Führen wir dies einmal an einem Beispiel durch. In Abbildung 8.11 sehen wir die 18 Register des 6845. In unserem Beispiel werden wir die Register R10 und R11 verwenden. Diese beiden Register bestimmen die erste und letzte Scan-Zeile für den Cursor. Auf der Farbkarte besteht nämlich jedes Zeichen aus acht Scan-Zeilen, die mit 0 bis 7 numeriert werden. Wir können nun den Cursor an einer beliebigen Stelle innerhalb dieser acht Zeilen aufbauen. Das Register R10 teilt dabei dem 6845 mit, mit welcher Scan-Zeile der Cursor beginnt, während das Register R11 die letzte Scan-Zeile des Cursors enhält. Das ROM BIOS setzt dabei seinerseits den Cursor auf die Zeilen 6 und 7. Dies wird dadurch erreicht, daß das Register R10 auf 6, das Register R11 auf 7 gesetzt wird.

Das Programm in Abbildung 8.12(a) verändert den Cursor auf der Farbkarte. Der Cursor wird dabei so umgestaltet, daß er in den ersten fünf Scan-Zeilen erscheint, anstelle wie bisher in den beiden unteren. Als erstes setzen wir dazu das CRT-Indexregister (3D4H) auf den Wert 10 und schreiben dann in das Datenregister (3D4H) den Startwert, nämlich 0. Sodann setzen wir das Indexregister auf 11 und schreiben die Nummer der letzten Scan-Zeile, nämlich 4. Der Cursor erscheint nun als blinkender Block in der oberen Hälfte jeder Zeichenposition, anstelle wie bisher als blinkende Unterstreichung. Ähnliche Techniken zur Modifikation des Cursors werden auch noch von etlichen anderen Editionsprogrammen für den PC verwendet, einschließlich zum Beispiel des Basic-Interpreters. Schalten wir dabei beispielsweise während des Editierens in den Einfügemodus, so wird der Cursor dicker. Der Basic-Interpreter bewirkt dies, indem er die Parameter für den 6845 verändert.

Gehen wir noch einmal zurück zu Abbildung 8.11, so können wir sehen, daß es noch viele andere zusätzliche Register im 6845 gibt. Die meisten von diesen Registern steuern die horizontalen und vertikalen Zeitsignale, die für ein Fernsehbild nötig sind. Wir können diese Werte verändern, um damit auch das Bild auf dem Bildschirm zu verändern. So verändert beispielsweise das MODE-Kommando im DOS, das den Bildschirm um eine Zeile nach oben verschieben kann, das R2-Register, das die Position des horizontalen Synchronimpulses beinhaltet.

Wollen Sie mit diesen Registern experimentieren, so sollten Sie kurze Programme schreiben, um die Veränderungen auszuführen. Sollten Sie versuchen, das DEBUG-Programm zu verwenden, um die Registerinhalte zu verändern, so erhalten Sie dabei kein Ergebnis. Die Register R14 und R15 dienen nämlich zur Steuerung der Cursorposition. Verändert die DEBUG-Routine diese Cursorposition, nachdem wir unsere Daten in das Indexregister bzw. das Datenregister des 6845 übertragen haben, so wird der von uns gewünschte Wechsel nicht auftreten. Dies rührt daher, daß die DEBUG-Routine ihrerseits den Inhalt von Index- und Datenregister des 6845 verändert hat, so daß diese nun nicht mehr den von uns eingesetzten Wert enthalten.

Von Interesse beim 6845 sind außerdem die Register R12 und R13, die die Startadresse enthalten. Die Farbkarte verfügt nämlich über 16K Speicher im Gegensatz zu nur 4K der Schwarz/Weiß-Karte. Bei der Farbkarte wird dieser zusätzliche Speicher für den Graphikmodus verwendet. Wir können ihn aber auch im Textmodus gebrauchen. Unser Text mit 80x25 Zeichen benötigt nämlich 4K Bytes, so daß wir im Bildspeicher vier verschiedene Bildschirmseiten aufbewahren können. Wie wir bereits in Abbildung 8.8 gesehen haben, können wir die Daten auf dem Bildschirm verschieben, indem wir sie einfach von einer Speicherstelle zur anderen transportieren. Ein vertikales Verschieben der Daten können wir allerdings auch ganz einfach dadurch erreichen, daß wir die Startadresse für den 6845 verändern. Normalerweise ist diese Startadresse 0. Ändern wir sie ab auf 80 (die Anzahl der Zeichen pro Zeile), so beginnt der Bildschirm mit dem ersten Zeichen der zweiten Zeile. Das Ergebnis ist ein scheinbares Hochschieben des Bildschirms um eine Zeile.

In Wirklichkeit haben wir dagegen die Daten nicht verschoben. Stattdessen haben wir den Bildausschnitt im Bildspeicher verschoben. Wir müssen uns dazu den 80x25 Zeichen großen Bildschirm als Fenster vorstellen, durch das wir in den 8192 Zeichen langen Bildschirmpuffer blicken können.

Verwenden wir nun die Startadresse des Bildschirmspeichers, um die Daten auf dem Bildschirm hin- und herzuschieben, tritt ein Problem auf, wenn wir an die Grenze des 16K großen Bildschirmpuffers gelangen. An diesem Punkt wird sich der Bildschirm nämlich umklappen. In diesem Fall kommen die oberen Zeilen des Bildschirms vom Ende des Bildschirmpuffers, während die unteren Zeilen des Bildschirms aus dem Anfangsbereich des Bildschirmpuffers entnommen werden. Wir werden sicher in der Lage sein, dieses Problem zu beherrschen, doch erfordert es einiges Nachdenken und auch einige Erfahrung.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.12 Color/Graphics Control Programs


 1                                          PAGE    ,132
 2                                          TITLE   Figure 8.12 Color/Graphics Control Programs
 3
 4       0000                       STACK   SEGMENT STACK
 5       0000      40 [                     DW      64 DUP(?)
 6                      ????
 7                            ]
 8
 9       0080                       STACK   ENDS
10
11       0000                       DISPLAY_BUFFER  SEGMENT AT 0B800H
12       0000                       DISPLAY_START   LABEL   WORD
13       0000                       DISPLAY_BUFFER  ENDS
14
15       = 03D4                     CRT_INDEX       EQU     03D4H
16       = 03D5                     CRT_DATA        EQU     03D5H
17       = 03DA                     CRT_STATUS      EQU     03DAH
18       = 000A                     CURSOR_START    EQU     10              ; Cursor registers
19       = 000B                     CURSOR_END      EQU     11              ;  within the 6845
20
21       0000                       CODE    SEGMENT
22                                          ASSUME  CS:CODE
23       0000                       COLOR_GRAPHICS  PROC    FAR
24       0000  1E                           PUSH    DS                      ; Set return address
25       0001  2B C0                        SUB     AX,AX
26       0003  50                           PUSH    AX
27
28                                  ;----- Figure 8.12(a) Modify cursor value
29
30       0004  BA 03D4                      MOV     DX,CRT_INDEX
31       0007  B0 0A                        MOV     AL,CURSOR_START         ; Point 6845 index register
32       0009  EE                           OUT     DX,AL                   ;  at the cursor start register
33       000A  42                           INC     DX
34       000B  B0 00                        MOV     AL,0
35       000D  EE                           OUT     DX,AL                   ; Set cursor start scan line
36
37       000E  4A                           DEC     DX
38       000F  B0 0B                        MOV     AL,CURSOR_END
39       0011  EE                           OUT     DX,AL                   ; Point at cursor end register
40       0012  42                           INC     DX
41       0013  B0 04                        MOV     AL,4
42       0015  EE                           OUT     DX,AL                   ; Set cursor end scan line
43
44                                  ;----- Figure 8.12(b) Use of status register
45
46       0016  B8 0002                      MOV     AX,2
47       0019  CD 10                        INT     10H                     ; Select 80x25 text mode
48       001B  B8  ---- R                   MOV     AX,DISPLAY_BUFFER
49       001E  8E C0                        MOV     ES,AX                   ; Address video buffer
50       0020  B8 0720                      MOV     AX,0720H                ; Start with " " character
51       0023                       NEXT_CHAR:
52       0023  BF 0000                      MOV     DI,0
53       0026  B9 0050                      MOV     CX,80
54       0029  F3/ AB                       REP     STOSW                   ; Write a line of 80 characters
55       002B  FE C0                        INC     AL                      ; Next character
56       002D  75 F4                        JNZ     NEXT_CHAR
57
58       002F  BB 0720                      MOV     BX,0720H
59       0032                       NEXT_CHAR_1:
60       0032  B9 0050                      MOV     CX,80
61       0035  BF 0000                      MOV     DI,0
62       0038  BA 03DA                      MOV     DX,CRT_STATUS           ; Status port for color card
63       003B                       WAIT_NO_RETRACE:
64       003B  EC                           IN      AL,DX
65       003C  A8 01                        TEST    AL,1                    ; Test retrace bit
66       003E  75 FB                        JNZ     WAIT_NO_RETRACE         ; Wait until it is displaying
67       0040  FA                           CLI                             ; Can't allow interrupts
68       0041                       WAIT_RETRACE:
69       0041  EC                           IN      AL,DX
70       0042  A8 01                        TEST    AL,1
71       0044  74 FB                        JZ      WAIT_RETRACE            ; Wait for non-display
72       0046  8B C3                        MOV     AX,BX                   ; Get character
73       0048  AB                           STOSW                           ; Store in buffer
74       0049  FB                           STI                             ; Interrupts back on
75       004A  E2 EF                        LOOP    WAIT_NO_RETRACE
76       004C  FE C3                        INC     BL
77       004E  75 E2                        JNZ     NEXT_CHAR_1
78
79                                  ;----- Figure 8.12(c) Draw a diágonal line in APA mode
80
81       0050  B8 0004                      MOV     AX,4
82       0053  CD 10                        INT     10H                     ; Select 320x200 APA mode
83
84       0055  06                           PUSH    ES
85       0056  1F                           POP     DS
```

```
86          0057  B3 32              MOV     BL,50              ; Number of row groups
87          0059  B1 02              MOV     CL,2               ; Shift count
88          005B  BF 0000            MOV     DI,0
89          005E               DOT_LOOP:
90          005E  B0 C0              MOV     AL,0C0H            ; First dot in byte
91          0060  88 05              MOV     [DI],AL
92          0062  81 C7 2000         ADD     DI,2000H           ; Move to odd row
93          0066  D2 E8              SHR     AL,CL              ; Shift to next pel in byte
94          0068  88 05              MOV     [DI],AL
95          006A  81 EF 1FB0         SUB     DI,2000H-80        ; Move to even row
96          006E  D2 E8              SHR     AL,CL
97          0070  88 05              MOV     [DI],AL
98          0072  81 C7 2000         ADD     DI,2000H           ; Move to odd row
99          0076  D2 E8              SHR     AL,CL
100         0078  88 05              MOV     [DI],AL
101         007A  81 EF 1FAF         SUB     DI,2000H-81        ; Move to even row, one byte over
102         007E  FE CB              DEC     BL
103         0080  75 DC              JNZ     DOT_LOOP
104
105                            ;----- Restore system to 80x25 text and return
106
107         0082  B8 0002            MOV     AX,2
108         0085  CD 10              INT     10H
109         0087  CB                 RET
110         0088               COLOR_GRAPHICS  ENDP
111         0088               CODE     ENDS
112                                  END
```

Abbildung 8.12 Steuerprogramm für die Farbkarte

Die Farbkarte verfügt über drei zusätzliche Ein/Ausgaberegister. Das Modusregister befindet sich dabei an der Ein/Ausgabeadresse 03D8H. Dieses Register bestimmt

die Steuerung der Hardware für die verschiedenen Arten der Datenausgabe. Beispielsweise können wir das Bit 5 dieses Registers auf 0 setzen und damit 16 verschiedene Hintergrundfarben im Textmodus ermöglichen. Setzen wir Bit 5 auf 1, so wählen wir damit die Blinkfunktion aus. Eine genaue Darstellung dieses Registers, wie auch aller anderen, finden Sie im Technical Reference Manual.

Das Farbwahlregister befindet sich an der Ein/Ausgabeadresse 3D9H. Dieses Register bestimmt im Textmodus die Farbe des Bildschirmrands. Da der ausgegebene Text nämlich nicht automatisch den ganzen Schirm belegt, können wir ihn mit einer gewählten Farbe umgeben. Das Farbwahlregister dient außerdem zur Auswahl von verschiedenen Farbpaletten im Graphikmodus, wie wir sie im nächsten Abschnitt besprechen werden. Die vier niedrigwertigen Bits des Registers dienen zur Auswahl von einer von 16 möglichen Farben, wie in Abbildung 8.10 gezeigt, für die Umrahmung des ausgegebenen Textes.

Schließlich übergibt noch das Statusregister an der Ein/Ausgabeadresse 3DAH relevante Informationen von der Farbkarte zurück in unser Programm. In einem Programm benötigen wir dieses Register, um den aktuellen Zustand des Lichtstifts oder Lightpens festzustellen, soweit einer mit dem System verbunden ist. Wichtiger noch, das Statusregister teilt uns mit, wann es gefahrlos für uns ist, Daten in den Bildschirmpuffer zu schreiben oder solche aus ihm zu lesen.

Ohne näher auf technische Details einzugehen, können wir bemerken, daß das Lesen von Daten aus dem Bildschirmspeicher oder das Schreiben von solchen in den Bildschirmspeicher bei der Verwendung der Farbkarte zu „Schnee" auf dem Bildschirm führen kann, wenn wir es zur falschen Zeit tun. Dies geschieht allerdings nur im hochlauflösenden Textmodus, wenn wir einen Bildschirm mit 80x25 Zeichen benützen. Sie werden sich daran erinnern, daß wir bereits in Abbildung 6.12 ein

Beispiel vorführten, bei dem fortlaufend ein Wert in den Bildschirmpuffer geschrieben wurde. Hätten wir bei diesem Beispiel die Farbkarte verwendet, so hätten wir auf dem Bildschirm eine Menge Störungen erhalten. Wir können nun das Statusregister verwenden, um dieses Problem zu umgehen.

Es gibt nämlich Zeitpunkte, zu denen es völlig ungefährlich ist, Daten im Bildschirmpuffer zu handhaben. Während dieser Zeiten steht das Bit 0 des Statusregisters auf 1. Wir müssen also nur darauf warten, daß das Bit 0 1 wird, um Daten in den Bildschirmpuffer zu schreiben oder daraus lesen zu können. In Abbildung 8.12(b) sehen wir ein Programm, bei dem ein Zeichen auf den Bildschirm ausgegeben wird. In den ersten Zeilen des Beispiels werden dabei Daten ohne Benutzung des Statusregisters in den Bildschirmpuffer geschrieben. Das Programm gibt dabei die 80 Zeichen der ersten Zeile des Bildschirms 224 mal aus. Dies entspricht etwa dem neunmaligen Füllen des ganzen Bildschirms. Wenn wir nun unser Programm ablaufen lassen, sehen wir auf dem Bildschirm ein kurzes Blitzen, wenn die Daten geschrieben werden.

Im zweiten Teil dieses Programms (b) wird die gleiche Aktion wiederholt, dieses Mal aber unter Beachtung des Statusbits. Halten wir fest, daß unser Programm dabei das Statusbit in zwei Arten testet. Als erstes warten wir, bis das Statusbit eine 0 enthält. Sobald es dann auf 1 umschaltet, schreibt unser Programm die Daten. Dies geschieht deshalb, weil es nur ein sehr kurzes Zeitintervall gibt, in dem die Daten geschrieben werden können. Würden wir nur auf eine 1 testen, so erhalten wir möglicherweise das Statusbit genau in dem Augenblick, wo es unmittelbar vor dem Umsprung auf eine 0 steht. Der Prozessor kann dann die Daten nicht schnell genug in den Bildschirmpuffer schreiben, um mögliche Interferenzen zu verhindern. Über unsere Programmschleife stellen wir also sicher, daß wir genau den Zeitpunkt erhalten, in dem das Statusbit das erste Mal auf 1 umschaltet.

Durch die Verwendung des Statusbits werden wir nun keinerlei Störungen mehr auf dem Bildschirm sehen. Wir werden allerdings feststellen, daß unser Programm zur Ausführung wesentlich mehr Zeit benötigt. Die zusätzliche Zeit, die nämlich dazu verwendet wird, auf das Statusbit zu warten, macht auch unser Programm langsamer. Wollen wir nun eine große Menge von Daten auf den Bildschirm schreiben — beispielsweise, das horizontale Verschieben der Daten von Abbildung 8.8 — werden wir eine andere Methode verwenden müssen. Der einfachste Weg dazu ist, den Bildschirm über das Modusregister einfach abzuschalten. Dazu gibt es im Mode-Register (3D8H) ein Bit (Bit 3), das die Bildschirmwiedergabe steuert. Setzen wir dieses Bit auf 0, so ist der Bildschirm leer. Wollen wir nun eine größere Datenmenge im Bildschirmpuffer hin- und herbewegen, so können wir ganz einfach den Bildschirm abschalten und die Daten dann ohne Testen des Statusbits hin- und hertransportieren. Da der Bildschirm während dieser Zeit ausgeschaltet ist, werden wir auf ihm auch keine Störungen sehen. Ist der Datenblock dann vollständig transportiert, schalten wir den Bildschirm wieder ein. Von außen werden wir dann nur ein kurzes Blinken sehen, wenn der Bildschirm zuerst aus- und dann wieder eingeschaltet wird. IBM verwendete diese Methode beispielsweise selbst zum vertikalen Datenverschieben auf dem Farbbildschirm.

Graphikmodus

Die Farbkarte verfügt über zwei Ausgabearten, bei denen jeweils jeder einzelne Punkt auf dem Bildschirm angesprochen werden kann. Wir bezeichnen diese Verarbeitungsart dann als „All Points Addressable" (APA), da wir jeden Punkt adressieren und verändern können. In Wirklichkeit gestattet es uns die Farbkarte sogar, auch andere Verarbeitungsarten als die beiden genannten APA-Modi anzuwenden, wir werden uns aber nur mit diesen beiden beschäftigen, da sie auch vom Programmiersystem unterstützt werden. Mit den Informationen, die Ihnen im Technical Reference Manual zur Verfügung stehen, können Sie auch die anderen zusätzlichen Verarbeitungsarten versuchen.

Im mittelauflösenden Graphikmodus verfügt der Bildschirm über 200 Zeilen zu je 320 Punkten. Jeder dieser Punkte kann dabei eine von vier möglichen Farben annehmen. Das bedeutet, daß wir für jeden Punkt zwei Bits benötigen, um die vier Farben darzustellen. Mit einem Byte im Graphikbildspeicher können wir also vier Bildpunkte oder Pixels (für „picture element") ansprechen. Multiplizieren wir die horizontalen mit den vertikalen Abmessungen, und dividieren wir das Ergebnis durch 4 für die Anzahl der Bildpunkte pro Byte, so werden wir sehen, daß wir im mittelauflösenden Graphikmodus 16.000 Bytes benötigen. Und dies ist auch der Grund, warum die Farbkarte über 16K Byte Speicher verfügt.

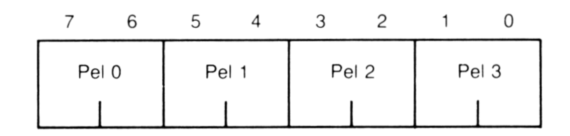


Abbildung 8.13 Bitbelegung für 320 x 200 Graphik

In Abbildung 8.13 sehen wir die Anordnung der einzelnen Pixels innerhalb eines einzelnen Bytes. Das Bit-Paar an den Stellen 7 und 6 wird dabei der erste darstellbare Bildpunkt. Die Bits 1 und 0 bilden dann den letzten Punkt, der mit diesem Byte dargestellt werden kann. Die ersten 80 Bytes des Graphikspeichers enthalten also die 320 Bildpunkte der Zeile 0, der ersten Zeile des Bildschirms.

Allerdings enthält nun das Byte 80 nicht die ersten vier Bildpunkte der Zeile 1. Aus Hardwaregründen werden die Zeilen mit geraden bzw. ungeraden Nummern an verschiedenen Stellen des Graphikspeichers aufbewahrt. Alle Zeilen mit ungeraden Nummern haben dabei einen Adressoffset von 2000H zur entsprechenden Zeile mit einer geraden Nummer. So befindet sich die Zeile 1 an den Bytes 2000H bis 204FH. Zeile 2, eine Zeile mit geradzahliger Nummer, befindet sich dann an den Bytes 50H bis 9FH. Zeile 3 befindet sich an den Stellen 2050H bis 209FH usw.

Abbildung 8.12(c) enthält ein Programm, das die Belegung des Graphikspeichers zeigt. Das Programm zieht dabei eine diagonal verlaufende Linie auf dem 320 × 200 Punkte großen Bildschirm. Die Linie bewegt sich von Punkt (0,0), dem linken oberen Eck, zum Punkt (199,199), etwas rechts von der Mitte am unteren

Rand des Bildschirms. Im ersten Teil des Programms wird dabei der Bildschirm in den mittelauflösenden APA-Modus mittels des ROM BIOS versetzt. Im nächsten Kapitel werden wir sehen, wie dies geschieht.

Wir verwenden nun das DI-Register, um auf das korrekte Byte im Bildschirmpuffer zu zeigen. Unser Programm gibt insgesamt 200 Punkte aus. Register BL wird auf 50 gesetzt, denn die innere Schleife schreibt jeweils 4 Bildpunkte in einem Durchgang. Außerdem initialisieren wir das Register CL mit 2, dem Shiftzähler. Die Schleife setzt nun das Register AL auf den Wert 0C0H, womit die beiden höchstwertigen Bits für die Farbe 3 gesetzt sind. Die restlichen 3 Pixels bleiben auf 0, der Hintergrundfarbe. Nachdem dieser Bildpunkt gespeichert ist, wird das Register AL um zwei Bits nach rechts verschoben, so daß wir nun das zweite Pixel im Byte ansprechen könnten. Eine Addition von 2000H auf das DI-Register läßt den Pointer nun auf die Zeile mit einer ungeraden Nummer zeigen. Der dritte Punkt in der inneren Schleife wird wiederum dadurch erreicht, daß wir das Register AL um 2 Bits nach rechts verschieben und nun den Wert (2000H-80) vom DI-Register subtrahieren. Das führt dazu, daß das DI-Register nun wieder auf eine Zeile mit einer geraden Nummer zeigt, in unserem Fall die Zeile 2 und außerdem auf die nächste, 80 Byte lange Dateneinheit. Haben wir nun mit unserer Schleife schließlich das vierte Byte gespeichert, so wird das DI-Register wiederum so gesetzt, daß es auf eine Zeile mit gerader Nummer zeigt, doch außerdem zusätzlich so erhöht, daß es auf das nächste Byte in dieser Zeile zeigt. Wir bearbeiten dabei absichtlich jeweils 4 Bytes innerhalb einer Schleife, da sich jeweils 4 Pixels in einem Byte befinden.

Farbdarstellung im 320x200 APA-Modus

Befassen wir uns nun einmal mit den Farben, über die wir im mittelauflösenden Graphikmodus verfügen können. Zwei Bits pro Pixel lassen uns die Wahlmöglichkeit zwischen vier verschiedenen Farben für jeden Bildpunkt, wobei die Farbe 0 (00B) die Hintergrundfarbe ist. Wir können allerdings jede beliebige Farbe aus den 16 möglichen, die in Abbildung 8.10 gezeigt sind, als Hintergrundfarbe wählen. Dazu speichern wir diesen 4-Bit-Wert im Farbwahlregister (3D9H). Die restlichen drei Farben sind dann von IBM bereits vordefiniert. Wir können die restlichen Farben 1, 2 und 3 also nicht beliebig auswählen. IBM stellt uns dazu zwei verschiedene Farbpaletten zur Verfügung, die wir in Abbildung 8.14 sehen. Die gewünschte Palette geben wir durch Bit 5 im Farbwahlregister an.

Wie wir in Abbildung 8.14 sehen, stehen uns die Farben grün, rot und gelb zur Verfügung, wenn wir Bit 5 auf 0 setzen, zusammen mit einer gewünschten Hintergrundfarbe. Das Setzen des Bit 5 auf 1 ergibt dann die Farben cyan, magenta und weiß. Mit Hilfe eines weiteren Bits im Farbwahlregister können wir allerdings auch noch diese Palettenwerte verändern. Durch Setzen des Bits 4 auf 1 erhalten wir die in der Palette enthaltenen Farben mit hoher Intensität. Die Initialisierungsroutinen des ROM BIOS setzen das Farbwahlregister normalerweise auf den Wert 30H. Dadurch wird die Hintergrundfarbe schwarz (0) und die Palette 1 mit hoher Intensität ausgewählt.

Farbwert	Farbpalette 0	Farbpalette 1
1 (01B)	grün	cyan
2 (10B)	rot	magenta
3 (11B)	gelb	weiß
	Bit 5 = 0	Bit 5 = 1

Abbildung 8.14 Farbpalette für 320x200 Graphik

Graphik mit hoher Auflösung

Der zweite verfügbare APA-Modus erlaubt uns eine Bildauflösung von 200 Zeilen zu je 640 Punkten. In diesem Modus verfügen wir nur über ein einziges Bit pro Pixel. Steht dieses Bit auf 0, so wird die entsprechende Stelle schwarz ausgegeben. Ist dieses Bit auf 1 gesetzt, wird die gewünschte Farbe ausgegeben. Die entsprechende Farbe wählen wir dabei mit dem Farbwahlregister. Normalerweise ist dieses Register auf den Wert 1111B (weiß) gesetzt, womit wir eine Schwarz/Weiß-Darstellung erhalten. Es steht jedoch in unserem Ermessen, jede beliebige andere Farbe zu wählen.

Die Belegung der Pixels entspricht dabei fast völlig dem Graphikmodus mit mittlerer Auflösung. Die einzige Ausnahme besteht darin, daß hier acht Bildpunkte mit einem Byte dargestellt werden. Das höchstwertige Bit (Bit 7) wird dabei als erstes ausgegeben, 0 als letztes. Auch hier verfügen wir wieder über 80 Bytes pro Zeile. Und genau wie beim Graphikmodus mit mittlerer Auflösung befinden sich auch hier die Zeilen mit geraden Nummern am Anfang des Bildschirmpuffers. Die Zeilen mit ungeraden Nummern beginnen dagegen bei der Adresse 2000H.

Paralleler Druckeradapter

Wir benötigen den parallelen Druckeradapter oder Druckerkarte, um den IBM-Drucker oder jeden beliebigen anderen Drucker, der über ein Parallelinterface angeschlossen wird, zu betreiben. Dieser Adapter ist auf der Schwarz/Weiß- und Druckerkarte bereits enthalten. Besitzen Sie jedoch eine Farbkarte, so benötigen Sie eine zusätzliche parallele Druckerkarte. Mit Ausnahme der Ein/Ausgabeadressen sind die beiden Adapterkarten dabei hinsichtlich des Druckerinterfaces vollkommen identisch. Auf der Schwarz/Weiß-Karte belegen die Druckerports die Adressen 3BCH bis 3BEH, auf der eigenständigen Druckerkarte dagegen die Adressen 378H bis 37AH.

Die Druckerkarte verfügt über zwei Ausgangsports und einen Eingangsport. Sie ähnelt dadurch sehr stark dem 8255, der für das Tastaturinterface verwendet wird. Und in der Tat war in den ursprünglichen Plänen auch der 8255 dafür vorgesehen. Allerdings entschloß sich IBM später, den Adapter eigenständig aufzubauen.

Der acht Bit breite Ausgabeport an der Adresse 3BCH bzw. 378H übergibt die Daten an den Drucker. Der vom Rechner in diesem Port übergebene ASCII-Zeichenwert wird vom Adapter dabei an den Drucker weitergeleitet. Der zweite Ausgabeport verfügt dagegen nur über fünf Ausgangsbits. Seine Adresse ist 3BEH bzw. 37AH. Dieser Port übermittelt die Steuersignale an den Drucker. Dadurch werden die Initialisierung und die Arbeitsweise des Druckers gesteuert. Im besonderen dient dabei Bit 0 zur Datenübergabe an den Drucker. Das Speichern der auszugebenden Zeichen im Ausgabeport bewirkt nämlich noch nicht, daß diese ASCII-Zeichen auch direkt an den Drucker gesendet werden. Dazu muß zunächst das Strobe-Bit (Bit 0 der Adressen 3BEH bzw. 37AH) auf 1 und danach wieder auf 0 gesetzt werden, damit der Drucker die Daten auch wirklich erhält. In Abbildung 8.15 sehen wir ein kurzes Programm, das einen Zeichenstring an den Drucker übergibt. Dieses Unterprogramm, das mit PRINT bezeichnet wird, übergibt die Daten tatsächlich an den Drucker.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.15 Printer Output

 1                                            PAGE    ,132
 2                                            TITLE   Figure 8.15 Printer Output
 3
 4        0000                        STACK   SEGMENT STACK
 5        0000      40 [                      DW      64 DUP(?)
 6                          ????
 7                               ]
 8
 9        0080                        STACK   ENDS
10
11        = 0378                      BASE    EQU     378H
12
13        0000                        CODE    SEGMENT
14                                            ASSUME  CS:CODE
15
16        0000  46 69 67 75 72 65     MSG     DB      'Figure 8.15',13,10,'$'
17              20 38 2E 31 35 0D
18              0A 24
19
20        000E                        MAIN    PROC    FAR
21        000E  1E                            PUSH    DS              ; Set return address
22        000F  2B C0                         SUB     AX,AX
23        0011  50                            PUSH    AX
24
25        0012  2E: 8D 1E 0000 R              LEA     BX,MSG
26        0017                        PRINT_LOOP:
27        0017  2E: 8A 07                     MOV     AL,CS:[BX]      ; Get character string byte
28        001A  3C 24                         CMP     AL,'$'          ; Is this the end of the string
29        001C  74 06                         JE      MAIN_RETURN
30        001E  E8 0025 R                     CALL    PRINT           ; Print the character
31        0021  43                            INC     BX
32        0022  EB F3                         JMP     PRINT_LOOP      ; Do the next character
33        0024                        MAIN_RETURN:
34        0024  CB                            RET
35        0025                        MAIN    ENDP
36
37                                    ;----- This routine prints the character in AL
38
39        0025                        PRINT   PROC    NEAR
40        0025  BA 0378                       MOV     DX,BASE       ; Data output port
41        0028  EE                            OUT     DX,AL         ; Put the character in the output port
42        0029  42                            INC     DX            ; Status port address
43        002A                        WAIT_BUSY:
44        002A  EC                            IN      AL,DX         ; Get the status
45        002B  A8 80                         TEST    AL,80H        ; Test the busy bit
46        002D  74 FB                         JZ      WAIT_BUSY     ; Loop if it's still busy
47        002F  42                            INC     DX            ; Point to control port
48        0030  B0 0D                         MOV     AL,0DH        ; Control value for strobe high
49        0032  EE                            OUT     DX,AL
50        0033  B0 0C                         MOV     AL,0CH        ; Control value for strobe low
51        0035  EE                            OUT     DX,AL
52        0036  C3                            RET
53        0037                        PRINT   ENDP
54        0037                        CODE    ENDS
55                                            END     MAIN
```

Abbildung 8.15 Druckerausgabe

Halten wir dabei fest, daß das Unterprogramm PRINT einen Wert vom Inputport (3BCH bzw. 379H) liest. Der Inputport dient zur Übergabe der Statusinformation vom Drucker zum Programm. Unser Beispielprogramm überprüft dabei, ob der Drucker bereit ist, das nächste Zeichen zu übernehmen. Bit 7 des Statusports gibt an, daß der Drucker gerade beschäftigt ist. Steht dieses Bit auf 1, so kann der Drucker ein weiteres Zeichen zum Drucken annehmen. Anderenfalls muß unser Programm warten. Die restlichen vier Bits des Inputports geben nur einige Fehlerbedingungen wieder, die während des Druckens auftreten könnten, wie z. B. Papierende. In unserem Beispiel überprüfen wir nicht auf diese Bedingungen. Das Technical Reference Manual beschreibt die Belegung der einzelnen Bits für Input- und Outputports auf der Druckerkarte.

Eines der Steuerbits an Port 3BEH bzw. 37AH kontrolliert die Drucker-Interruptleitung. Dieses Bit muß auf 1 gesetzt sein, bevor der Drucker seinen Interrupt zurück an den 8259 auf der Systemplatine senden kann. Allerdings müssen wir hier sagen, daß der Druckeradapter dazu die falsche Leitung benützt. Das gewählte Signal ergibt nämlich keine Unterbrechung. Sie sollten deshalb keine Programme schreiben, die die Interruptfähigkeit der Druckerkarte benützen (außer, Sie wollen Ihre Druckerkarte selbst abändern). Wir werden noch an einem Beispiel zeigen, wie man dieses Problem durch Verwendung des Systemzeitgebers umgehen kann.

Asynchronadapter

Die Asynchronkarte (Asynchronous Communications Adapter) gibt dem IBM PC die Möglichkeit, über eine serielle Schnittstelle zu verfügen. Diese Karte gibt uns die Möglichkeit, mit anderen Rechnern, Datenbasen-Systemen und anderen Informationsanbietern zu verkehren. Wir wollen hier jedoch nicht besprechen, wie die asynchrone Datenübertragung ganz allgemein abläuft, sondern nur die Methode, wie wir diesen besondern Adapter für den IBM PC programmieren können.

Die gesamte Arbeit des Sendens und Empfangens von Daten über eine asynchrone Verbindung bewirkt dabei ein einziger Kommunikationsschaltkreis. Wir können diesen Baustein, das 8250 Asynchronous Communication Element (ACE), so programmieren, daß es eine Vielzahl von Datenübertragungsmöglichkeiten beherrscht. Die Zeichengröße, Übertragungsrate, Stoppzeichen und Parity-Bits können alle von unserem Programm bestimmt werden, wenn wir das ACE initialisieren. Außerdem gestattet uns der Adapter, die normalen Modem-Steuersignale (Modulator-Demodulator) zu überprüfen.

Wir senden Daten über das ACE, indem wir das entsprechende Zeichen in das Senderegister ablegen. Der Baustein bewerkstelligt dann alles andere für uns, entsprechend den Werten, mit denen wir ihn initialisiert haben. Um ein Zeichen zu empfangen, müssen wir es nur aus dem Empfangspuffer lesen. Außerdem gibt es ein Statusregister — das Leitungszustandsregister — das uns anzeigt, wann der Sendepuffer leer ist und ein weiteres Zeichen aufnehmen kann. Ein weiteres Bit im Statusregister teilt uns mit, wann das ACE ein Zeichen von einem anderen System empfangen hat.

Das Technical Reference Manual zeigt uns auch noch die anderen Register, die Teil des 8250 ACE sind. Diese Register dienen unter anderem dazu, das Modem zu steuern und seinen jeweiligen Zustand festzustellen. Wir können außerdem das ACE dazu veranlassen, daß verschiedene Bedingungen einen Interrupt erzeugen. Dies gestattet es unserem Programm dann, schnell auf eine entsprechende Änderung der externen Bedingungen zu reagieren.

```
The IBM Personal Computer MACRO Assembler 01-01-83       PAGE     1-1
Figure 8.16 Asynchronous setup, send and receive

1                                               PAGE    ,132
2                                               TITLE   Figure 8.16 Asynchronous setup, send and receive
3
4       0000                            STACK   SEGMENT STACK
5       0000      40 [                          DW      64 DUP(?)
6                       ????
7                             ]
8
9       0080                            STACK   ENDS
10
11      = 03F8                          SERIAL  EQU     03F8H
12
13      0000                            CODE    SEGMENT
14                                              ASSUME  CS:CODE
15      0000                            ASYNC   PROC    FAR
16      0000  52                                PUSH    DX
17      0001  2B C0                             SUB     AX,AX
18      0003  50                                PUSH    AX
19      0004  BA 03FB                           MOV     DX,SERIAL+3             ; Control register
20      0007  B0 80                             MOV     AL,80H
21      0009  EE                                OUT     DX,AL                   ; Setup for divisor value
22      000A  B8 0180                           MOV     AX,384                  ; Divisor for 300 baud
23      000D  BA 03F8                           MOV     DX,SERIAL
24      0010  EE                                OUT     DX,AL                   ; Low order of divisor
25      0011  8A C4                             MOV     AL,AH
26      0013  42                                INC     DX
27      0014  EE                                OUT     DX,AL                   ; High order of divisor
28      0015  BA 03FB                           MOV     DX,SERIAL+3             ; Control register
29      0018  B0 03                             MOV     AL,00000011B            ; No parity, 8 bit characters
30      001A  EE                                OUT     DX,AL
31
32                                      ;----- This section will send a character
33
34      001B  BA 03FD                           MOV     DX,SERIAL+5             ; Line status register
35      001E                            SEND:
36      001E  EC                                IN      AL,DX
37      001F  A8 20                             TEST    AL,20H
38      0021  74 FB                             JZ      SEND
39      0023  B0 41                             MOV     AL,'A'
40      0025  BA 03F8                           MOV     DX,SERIAL
41      0028  EE                                OUT     DX,AL
42
43                                      ;----- This section will receive a character
44
45      0029  BA 03FD                           MOV     DX,SERIAL+5             ; Line status register
46      002C                            RECV:
47      002C  EC                                IN      AL,DX
48      002D  A8 02                             TEST    AL,2
49      002F  74 FB                             JZ      RECV
50      0031  BA 03F8                           MOV     DX,SERIAL
51      0034  EC                                IN      AL,DX
52
53      0035  CB                                RET
54      0036                            ASYNC   ENDP
55      0036                            CODE    ENDS
56                                              END     ASYNC
```

Abbildung 8.16 Asynchronbetrieb: Vorbereitung, Senden, Empfang

Das Programm in Abbildung 8.16 zeigt Ihnen die grundlegenden Vorgänge, die notwendig sind, um das ACE zu initialisieren, ein Zeichen zu senden oder ein anderes Zeichen zu empfangen. Die Haupt-Ein/Ausgabeadresse der Adapterkarte ist 3F8H, so daß die einzelnen Register des ACE die Adressen 3F8H bis 3FEH belegen. Wir können jedoch die IBM-Asynchronkarte so modifizieren, daß sie auch auf die Ein/Ausgabeadressen 2F8H bis 2FEH antwortet. Auf diese Weise können wir zwei getrennte Asynchronkarten im IBM PC installieren und mit zwei verschiedenen externen Geräten kommunizieren. Es ist also in der Tat auch möglich, einen Drucker an das System über eine serielle Schnittstelle anstelle einer parallelen anzuschlie-

ßen. In diesem Fall benötigen wir dann zwei Asynchronkarten. Die eine Karte steuert den Drucker und die andere behandelt die übrigen externen Kommunikationen.

Eine der Ein/Ausgabeadressen des ACE übernimmt dabei verschiedene Aufgaben. Der Sende- und Empfangspuffer befinden sich nämlich beide an der Ein/Ausgabeadresse 3F8H. Schreiben wir an diese Stelle, so handelt es sich um den Sendepuffer, lesen wir jedoch von ihr, so erhalten wir das letzte Zeichen, das vom ACE empfangen wurde. Dieser Ein/Ausgabeport hat außerdem noch eine dritte Aufgabe. Der Divisorwert, der die Übertragungsrate für den Adapter bestimmt, wird in diesem Ein/Ausgabeport abgelegt. Das ACE dividiert dann die Taktfrequenz durch diesen Wert und erlaubt es uns so, eine Übertragungsrate zwischen 50 und 9600 auszuwählen. Ein Bit im Steuerregister bestimmt dabei die Verwendung des Ports 3F8H.

Im ersten Teil unseres Beispielprogramms wird das 8250 ACE initialisiert. Die erste Aktion unseres Programms besteht darin, die Übertragungsrate für den Adapter festzulegen. Ein Divisorwert von 384 ergibt dabei die gewünschte Baudrate von 300. Halten wir dabei fest, daß unser Programm das Bit 7 des Ports 3FBH, des Steuerregisters, auf 1 setzt, bevor der Divisorwert gespeichert wird. Der letzte Ausgabebefehl an den Port 3FBH bestimmt schließlich noch die Charakteristik der Datenübertragung. In unserem Beispiel wählten wir dazu 8-Bit-Zeichen ohne Parity.

In den verbleibenden beiden Abschnitten unseres Programms senden bzw. empfangen wir ein Zeichen. Das Leitungsstatusregister im Port 3FBH hat dabei jeweils ein Statusbit für den Sendepuffer und ein zweites für den Empfangspuffer. Wir können nämlich kein weiteres Zeichen senden, bevor nicht der Sendepuffer wieder leer ist. Und wir können ganz sicherlich kein neues Zeichen lesen, bevor der Adapter es nicht empfangen hat.

Die Asynchronkarte unterstützt außerdem Interrupts. Das OUT2-Signal des Modem-Steuerregisters verbindet das Interruptsignal des ACE mit dem System. Das Interrupt-Enable-Register im ACE wählt dabei aus, welche der möglichen Zustandsänderungen eine externe Unterbrechung erzeugen soll. Die Asynchronkarte erzeugt dabei Interrupts der Ebene 3 für den 8259.

Sehen wir uns nun einmal an, was nötig ist, um die Asynchronkarte zur Erzeugung der Interrupts zu veranlassen, die wir zum Empfang eines Zeichens benötigen. In Abbildung 8.17 sehen wir, was nötig ist, um das Interruptsystem einzuschalten. Für den Hardwareinterrupt setzt unser Programm dabei den Interruptvektor für Ebene 3 des 8259 (Interrupt 0BH an der Stelle 58H), um die entsprechende Interruptserviceroutine zu adressieren. Im Interruptmaskenregister wird sodann an der Stelle, die dem Interrupt von der Asynchronkarte entspricht, das Bit gelöscht. Im 8250 ACE wird nun seinerseits das Interrupt-Enable-Register so gesetzt, daß Leitungsunterbrechungen an der Empfangsleitung ermöglicht sind. Schließlich schaltet unser Programm noch die OUT2-Leitung ein, um das Senden der Interrupts an das System zu ermöglichen. Nach alldem ist es nun kein Problem mehr, die Zeichen zu verarbeiten, wenn sie vom System empfangen werden. In unserem Beispiel werden sie in einem Puffer abgelegt, wo ein anderes Programm ganz nach Belieben auf sie zugreifen kann.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.17  Asynchronous Interrupts

1                                              PAGE    ,132
2                                              TITLE   Figure 8.17  Asynchronous Interrupts
3       0000                            ABS0    SEGMENT AT 0
4       002C                                    ORG     0BH*4
5       002C                            ASYNC_INTERRUPT LABEL   WORD
6       002C                            ABS0    ENDS
7       0000                            STACK   SEGMENT STACK
8       0000       40 [                         DW      64 DUP(?)
9                        ????
10                            ]
11
12      0080                            STACK   ENDS
13
14      0000                            CODE    SEGMENT
15                                              ASSUME  CS:CODE
16      0000  0049 R                    BUFFER_POINTER  DW      BUFFER
17
18      0002                            SET_INTERRUPT   PROC    FAR
19      0002  2B C0                             SUB     AX,AX
20      0004  8E D8                             MOV     DS,AX
21                                              ASSUME  DS:ABS0         ; Addressing to interrupt vectors
22
23                                      ;----- Set up the interrupts on the 8250
24
25      0006  C7 06 002C R 0024 R               MOV     ASYNC_INTERRUPT,OFFSET INT_HANDLER
26      000C  8C 0E 002E R                      MOV     ASYNC_INTERRUPT+2,CS    ; Set the 8259 interrupt vector
27
28      0010  BA 03F9                           MOV     DX,03F9H        ; Interrupt Enable Register
29      0013  B0 04                             MOV     AL,04H          ; Receive line status interrupt
30      0015  EE                                OUT     DX,AL           ; Set the register
31
32      0016  E4 21                             IN      AL,21H          ; 8259 Interrupt mask register
33      0018  24 F7                             AND     AL,0F7H         ; 0 at bit 3
34      001A  E6 21                             OUT     21H,AL          ; Interrupt unmasked
35
36      001C  BA 03FC                           MOV     DX,3FCH         ; Modem control register
37      001F  B0 08                             MOV     AL,08H          ; OUT2 bit
38      0021  EE                                OUT     DX,AL           ; Interrupts enabled at async card
39
40      0022  EB FE                     HERE:   JMP     HERE            ; Program done, wait for interrupt
41      0024                            SET_INTERRUPT   ENDP
42
43                                      ;----- Interrupt handler for receive characters
44
45      0024                            INT_HANDLER     PROC    FAR
46      0024  50                                PUSH    AX
47      0025  53                                PUSH    BX              ; Save interrupted register
48      0026  52                                PUSH    DX
49      0027  BA 03FD                           MOV     DX,3FDH         ; Line status register
50      002A  EC                                IN      AL,DX
51      002B  A8 01                             TEST    AL,01H          ; Make sure a char was received
52      002D  74 12                             JZ      INT_RETURN      ; Something wrong, return
53      002F  BA 03F8                           MOV     DX,3F8H         ; Receive data register
54      0032  EC                                IN      AL,DX           ; Get input char
55      0033  2E: 8B 1E 0000 R                  MOV     BX,BUFFER_POINTER
56      0038  2E: 88 07                         MOV     CS:[BX],AL      ; Store the input in buffer
57      003B  43                                INC     BX
58      003C  2E: 89 1E 0000 R                  MOV     BUFFER_POINTER,BX
59      0041                            INT_RETURN:
60      0041  5A                                POP     DX
61      0042  5B                                POP     BX              ; Restore registers
62      0043  B0 20                             MOV     AL,20H          ; EOI command
63      0045  E6 20                             OUT     20H,AL
64      0047  58                                POP     AX
65      0048  CF                                IRET                    ; Return from interrupt
66      0049                            INT_HANDLER     ENDP
67      0049       80 [                 BUFFER  DB      128 DUP(0)
68                       00
69                            ]
70
71      00C9                            CODE    ENDS
72                                              END     SET_INTERRUPT
```

Abbildung 8.17 Asynchrone Interrupts

Spieladapter

Der Spieladapter verbindet die Joysticks mit dem System. Diese sind Analoggeräte — das heißt, sie arbeiten nicht auf einer Basis von Einsen und Nullen. Der an einen Joystick angeschlossene Eingang ist dabei kein Binärwert, den ein Computer direkt lesen könnte, sondern ein Widerstandswert. Der Spieladapter konvertiert nun diesen Widerstandswert in etwas, womit der Computer arbeiten kann.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE     1-1
Figure 8.18  Game Control Adapter

1                                          PAGE    ,132
2                                          TITLE   Figure 8.18  Game Control Adapter
3
4        = 0201                    GAME_PORT       EQU     201H
5
6        0000                      STACK   SEGMENT STACK
7        0000      40 [                    DW      64 DUP(?)
8                       ????
9                             ]
10
11       0080                      STACK   ENDS
12
13       0000                      CODE    SEGMENT
14                                         ASSUME  CS:CODE
15       0000                      GAME_CONTROL    PROC    FAR
16       0000  1E                          PUSH    DS                  ; Set up return address
17       0001  2B C0                       SUB     AX,AX
18       0003  50                          PUSH    AX
19       0004  BA 0201                     MOV     DX,GAME_PORT
20       0007  B8 B000                     MOV     AX,0B000H           ; Display buffer segment
21       000A  8E D8                       MOV     DS,AX
22
23       000C  B5 21                       MOV     CH,21H              ; Character to write in buffer
24       000E  B1 00                       MOV     CL,0                ; Switch settings
25       0010                      WRITE_LOOP:
26       0010  B4 01                       MOV     AH,1                ; Get X position
27       0012  E8 0042 R                   CALL    POSITION
28       0015  8B D8                       MOV     BX,AX               ; Save X position
29       0017  D1 EB                       SHR     BX,1
30       0019  D1 EB                       SHR     BX,1                ; Divide by 4
31       001B  B4 02                       MOV     AH,2                ; Get Y position
32       001D  E8 0042 R                   CALL    POSITION
33       0020  D0 E8                       SHR     AL,1
34       0022  D0 E8                       SHR     AL,1
35       0024  D0 E8                       SHR     AL,1                ; Divide by 16
36       0026  D0 E8                       SHR     AL,1
37       0028  B4 A0                       MOV     AH,160
38       002A  F6 E4                       MUL     AH                  ; Convert to buffer offset
39       002C  03 D8                       ADD     BX,AX
40       002E  88 2F                       MOV     [BX],CH             ; Store a character there
41       0030  EC                          IN      AL,DX
42       0031  24 10                       AND     AL,10H
43       0033  3A C1                       CMP     AL,CL
44       0035  74 D9                       JE      WRITE_LOOP
45       0037  8A C8                       MOV     CL,AL
46       0039  80 F9 10                    CMP     CL,10H
47       003C  75 D2                       JNE     WRITE_LOOP
48       003E  FE C5                       INC     CH                  ; Next character
49       0040  EB CE                       JMP     WRITE_LOOP
50
51       0042                      GAME_CONTROL    ENDP
52
53                                 ;----- AH has mask bit
54
55       0042                      POSITION        PROC    NEAR
56       0042  51                          PUSH    CX
57       0043  2B C9                       SUB     CX,CX               ; Set loop control value
58       0045  EE                          OUT     DX,AL               ; Start the timing
59       0046                      POS_LOOP:
60       0046  EC                          IN      AL,DX
61       0047  84 C4                       TEST    AL,AH
62       0049  E0 FB                       LOOPNE  POS_LOOP            ; Loop while the bit is still on
63       004B  B8 0000                     MOV     AX,0
64       004E  2B C1                       SUB     AX,CX               ; Determine count value
65       0050  59                          POP     CX                  ; Range is 0 - 255
66       0051  C3                          RET
67       0052                      POSITION        ENDP
68       0052                      CODE    ENDS
69                                         END     GAME_CONTROL
```

Abbildung 8.18 Spieladapter

Dabei übersetzt der Spieladapter den Widerstandswert nicht direkt in einen Binärwert. Stattdessen wird der Widerstandswert vom Adapter in eine Zeitverzögerung umgesetzt. Je größer also der Widerstand ist, desto größer wird auch die Zeitverzögerung. Der Computer kann nun diese Zeitverzögerung messen. Und ein Programm kann diese Zeitverzögerung in eine Zahl verwandeln, die dann der Position des Joysticks entspricht. Unser Problem ist es nun, ein Programm zu schreiben, das diese Zeitverzögerung in eine Zahl verwandelt.

An den Spieladapter können bis zu vier Widerstandseingänge angeschlossen werden. Der Zeitverzögerungsmechanismus für jeden einzelnen dieser Eingänge ist mit einem eigenen Bit im Ein/Ausgabeport 201H verbunden. Geben wir nun irgendeinen beliebigen Wert auf den Port 201H aus, so werden die vier niedrigwertigen Bits des Ports 0. Nach Verstreichen einer bestimmten Zeit werden die einzelnen Bits 1. Und diese Zeit ist abhängig von dem Wert des jeweils angeschlossenen Widerstands. Das Programm in Abbildung 8.18 ist ein Beispiel dafür, wie wir die Widerstandswerte von zwei der vier Eingangsports erschließen können. Wir verwenden dabei eine sehr einfache Methode. Anstatt nämlich die Zeitverzögerung aller vier Eingänge gleichzeitig zu ermitteln, werden sie sequentiell behandelt. Die Zeit, die dabei für die Ermittlung eines einzelnen Werts verwendet wird, ist kurz. Deswegen ist es durchaus zulässig, die beiden Werte nacheinander abzuarbeiten als alle gleichzeitig.

Der letzte Teil des Programms in Abbildung 8.18 benützt nun die X-Y-Position, die wir durch den Eingang der Joysticks ermittelt haben, und schreibt ein Zeichen an die entsprechende Position auf dem Bildschirm. Der Spieladapter verfügt außerdem über vier Schalteingänge, deren Zustand als die höherwertigen vier Bits des Ports 201H gelesen werden können. Unser Beispielprogramm überprüft nun eines dieser Schaltbits, um einen Anhaltspunkt dafür zu haben, wann das Zeichen auf dem Bildschirm bewegt werden soll.

Diskettenadapter

Der Diskettenadapter ist das Interface zwischen dem Prozessor und den Diskettenlaufwerken. Die Schaltkreise auf der Karte beinhalten dabei alle notwendigen Funktionen zum Sichern bzw. Bereitstellen von Daten auf der Diskette. Der Adapter erledigt außerdem die physikalische Formatierung der Daten, die bei der Verwendung von Disketten nötig ist.

Das zentrale Bauelement auf dem IBM Diskettenadapter ist der NEC PD765 Floppy Disk Controller (FDC). Dieses Element wird auch von Intel unter der Bezeichnung 8272 geliefert. Der FDC steuert dabei den Datenfluß von und nach der Diskette. Der FDC verfügt dazu über zwei Ein/Ausgabeports, einen für Daten und einen für den Status. Der Datenport befindet sich an der Ein/Ausgabeadresse 3F4H, der Statusport an der Adresse 3F5H. Der Datenport ist dabei bidirektional. Das heißt, zu verschiedenen Zeiten können wir über diesen Ein/Ausgabeport die Daten sowohl

lesen als auch schreiben. Das Statusregister ist dagegen nur zum Lesen geeignet, es kann allerdings zu jeder Zeit gelesen werden. Das Statusregister teilt uns außerdem mit, wie wir jeweils den Datenport zu behandeln haben.

Es gibt zwei Bits im Statusport, die wir bei jeder Diskettenoperation benötigen. Bit 6 ist dabei der Daten-Input/Output (DIO). Dieses Bit teilt uns mit, ob der FDC Daten von uns erwartet. Ist das DIO-Bit auf 1 gesetzt, so erwartet der FDC von uns, daß wir aus dem Datenregister lesen. Ist das DIO-Bit auf 0 gesetzt, so erwartet der FDC von uns, daß wir irgendetwas in das Datenregister schreiben. Das Bit 7 des Statusports wird als „Request for Master"-Bit (RQM) bezeichnet. Dies entspricht in etwa dem Signal „busy" des Druckers. Ist dieses RQM-Bit auf 1 gesetzt, so ist der FDC für uns frei für eine Lese- oder Schreiboperation auf das Datenregister. Wenn wir allerdings das RQM-Bit nicht beachten, so kann es geschehen, daß wir den FDC verwirren, und dann funktioniert wirklich nichts mehr.

Das Datenregister ist in Wirklichkeit kein einzelnes Register. Ähnlich dem 6845 CRT besteht der Datenport aus einer Gruppe von Registern. Im Gegensatz zum 6845 aber gibt es hier keinen Index für das Datenregister. Die Daten, die wir an den Controller übergeben, müssen sich also in der richtigen Reihenfolge befinden. In ähnlicher Weise kommen die Informationen, wenn wir Daten von diesem Port lesen, in einer bestimmten Reihenfolge.

Das Technical Reference Manual enthält eine Tabelle, die uns alle möglichen Kombinationen von Eingabe- und Ausgabeoperationen für das Diskettenlaufwerk zeigt. Sehen wir uns einmal ein einfaches Kommando an den FDC, das Ermitteln des Gerätezustands, an. Wir führen diese Operation aus, wenn wir etwas über den aktuellen Zustand des Diskettenlaufwerks erfahren wollen. Im Abbildung 8.19 sehen wir die Daten, die für diesen Befehl notwendig sind. Und in Abbildung 8.20 sehen wir ein Programm, das diesen Befehl ausführt.

Befehl	**04H**
Befehlsmodifikation	**00H**
Status	**ST3**

Abbildung 8.19 Disketten-Statusbefehl

Jede Aktion des Diskettencontrollers hat drei Phasen: Befehl, Ausführung und Ergebnis. In der Befehlsphase erwartet der FDC Daten, und das DIO-Bit zeigt dieses an. Setzt der FDC nun das RQM-Bit, um uns anzuzeigen, daß er bereit ist, Daten zu akzeptieren, so kann unser Programm einen Befehl an den Controller senden. Zum Ermitteln des Gerätezustands geben wir beispielsweise zwei Befehlsbytes an den FDC. Das erste Byte, 04H, enthält den Operationscode für diese Statusoperation. Das zweite Byte teilt dem FDC mit, welches Diskettenlaufwerk er überprüfen soll. Während der Kommandophase zeigt das DIO-Bit immer an, daß der FDC auf Daten wartet, und unser Programm verwendet das RQM-Bit, um festzustellen, wann ein geeigneter Augenblick gekommen ist, um das Byte auszugeben.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.20 Sense Drive Status


1                                              PAGE    ,132
2                                              TITLE   Figure 8.20 Sense Drive Status
3
4       0000                           STACK   SEGMENT STACK
5       0000      40 [                         DW      64 DUP(?)
6                         ????
7                               ]
8
9       0080                           STACK   ENDS
10
11                                     FDC_STATUS      RECORD  RQM:1,DIO:1,OTHER:6
12
13      0000                           CODE    SEGMENT
14                                             ASSUME  CS:CODE
15      0000                           SENSE   PROC    FAR
16      0000  1E                               PUSH    DS              ; Set return address
17      0001  2B C0                            SUB     AX,AX
18      0003  50                               PUSH    AX
19
20      0004  BA 03F4                          MOV     DX,03F4H        ; Status port for FDC
21      0007  B4 04                            MOV     AH,04H          ; Sense drive status command
22      0009  E8 001E R                        CALL    OUTPUT          ; Send to controller
23      000C  B4 00                            MOV     AH,0            ; Second byte of command
24      000E  E8 001E R                        CALL    OUTPUT
25
26                                     ;----- Read the status from the FDC
27
28      0011                           IN_DIO:
29      0011  EC                               IN      AL,DX           ; Wait until DIO indicates
30      0012  A8 40                            TEST    AL,MASK DIO     ;  input from FDC
31      0014  74 FB                            JZ      IN_DIO
32      0016                           IN_RQM:
33      0016  EC                               IN      AL,DX           ; Wait until RQM indicates
34      0017  A8 80                            TEST    AL,MASK RQM     ;  FDC is ready
35      0019  74 FB                            JZ      IN_RQM
36
37      001B  42                               INC     DX              ; Point to data port
38      001C  EC                               IN      AL,DX           ; Read in the sense status
39      001D  CB                               RET                     ; End of example
40      001E                           SENSE   ENDP
41
42                                     ;----- Routine to send byte to FDC
43
44      001E                           OUTPUT  PROC    NEAR
45      001E  EC                               IN      AL,DX           ; Wait until DIO says FDC ready
46      001F  A8 40                            TEST    AL,MASK DIO     ;  for output to it
47      0021  75 FB                            JNZ     OUTPUT
48      0023                           OUT_RQM:
49      0023  EC                               IN      AL,DX           ; Wait until RQM says FDC ready
50      0024  A8 80                            TEST    AL,MASK RQM
51      0026  74 FB                            JZ      OUT_RQM
52
53      0028  42                               INC     DX              ; Point at data port
54      0029  8A C4                            MOV     AL,AH           ; Get data value
55      002B  EE                               OUT     DX,AL           ; Send the value
56      002C  4A                               DEC     DX              ; Back to status port
57      002D  C3                               RET
58      002E                           OUTPUT  ENDP
59      002E                           CODE    ENDS
60                                             END     SENSE
```

Abbildung 8.20 Disketten-Status

Der Controller tritt nun in die Ausführungsphase ein. Während dieser Phase wird der von uns übergebene Befehl ausgeführt. In unserem Beispiel wird dabei der Zustand des Geräts überprüft. Während dieser Aktion zeigt das RQM-Bit unserem Programm an, daß wir den Datenport besser in Ruhe lassen. Nach Ausführung des Befehls ändert sich der Zustand des DIO-Bits auf 1 und teilt unserem Programm dadurch mit, daß wir das Datenregister lesen sollen. Wenn es uns das RQM-Bit nun gestattet, kann unser Programm das einzelne Statusbyte, also das Ergebnis der Operation, lesen. Haben wir einmal die gesamte Statusinformation gelesen, so wechselt das DIO-Bit wieder zurück auf 0 und zeigt solchermaßen an, daß die Eingabe für den nächsten Befehl erwartet wird.

Wie wir aus der Tabelle im Technical Reference Manual sehen können, ist der Befehl zum Überprüfen des Gerätestatus einer der einfachsten. Der Befehl zum Datenlesen beispielsweise benötigt neun Datenbytes während der Befehlsphase. Ist der Befehl beendet, so muß das Programm sieben Statusbytes vom Controller lesen. Die Befehlsabarbeitung beginnt dabei nicht eher, als bis nicht das letzte Befehlsbyte gesendet ist, und wir können auch keinen weiteren Befehl übermitteln, bevor wir nicht alle sieben Statusbytes gelesen haben.

An der Ein/Ausgabeadresse 3F2H befindet sich ein zusätzliches, digitales Ausgaberegister des Diskettencontrollers. Dieser Ausgabeport erledigt einige zusätzliche Aufgaben zur Steuerung des Diskettenlaufwerks. Der hauptsächliche Verwendungszweck dieses Ports ist dabei die Steuerung des Diskettenmotors. In den 5 1/4-Zoll Diskettenlaufwerken des IBM PC läuft der Motor nämlich nicht ununterbrochen. Ein Programm muß also diesen Motor erst einschalten, bevor es Daten lesen oder schreiben kann, und ihn nach dieser Operation auch wieder ausschalten. Würden wir nämlich den Motor die ganze Zeit über laufen lassen, so würden wir die Disketten unnötigerweise abnützen. Ist der Motor eingeschaltet, so leuchtet auch das rote Licht an der Frontplatte des Diskettenlaufwerks.

Der Diskettenadapter verwendet dieses digitale Ausgangsregister allerdings auch noch für einige zusätzliche andere Funktionen. Zwei der Bits dienen dabei zur Auswahl des Diskettenlaufwerks. Das Register verfügt außerdem noch über eine Möglichkeit, den FDC zurückzusetzen, da es Fehlerbedingungen geben kann, die den Controller in einen undefinierbaren Zustand versetzen. In diesen Fällen ist dann die einzige Lösung, den Controller zurückzusetzen und die Operation erneut zu versuchen.

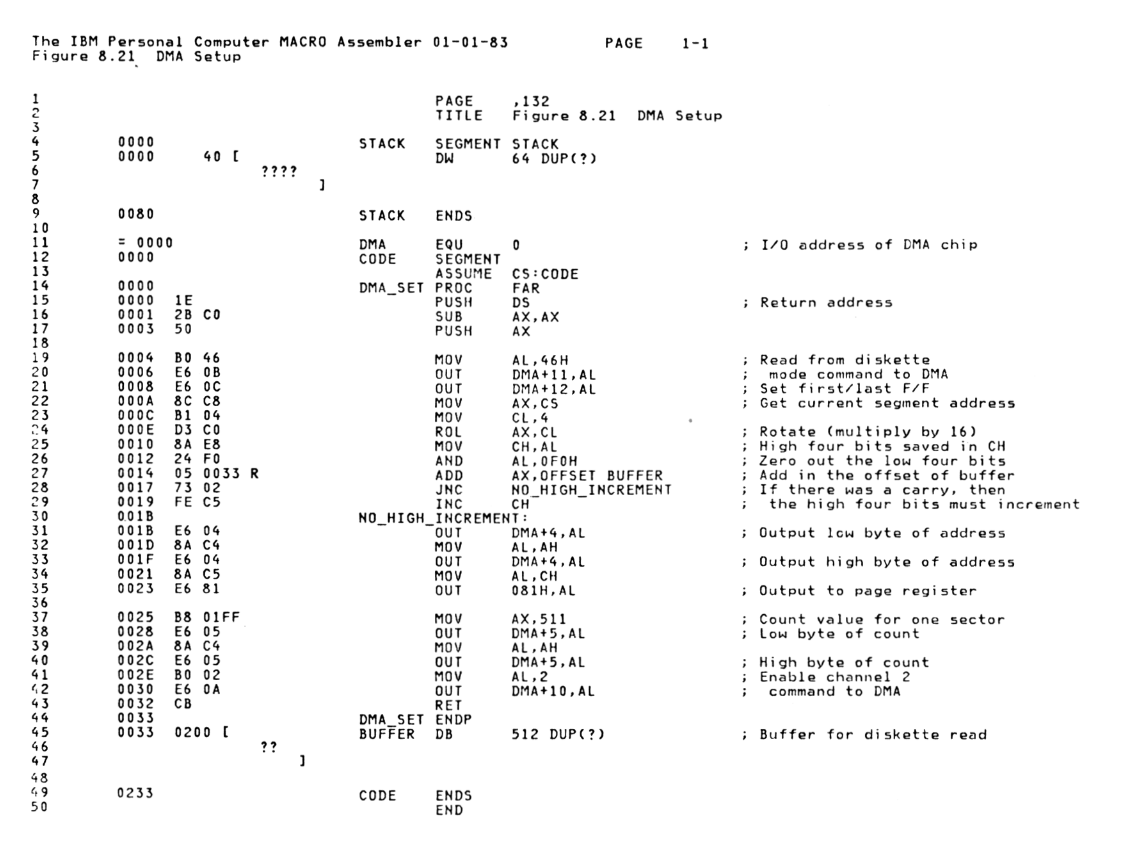
Direkter Speicherzugriff

Der Diskettenadapter ist eine IBM-Platine, die die Möglichkeit des direkten Speicherzugriffs — Direct Memory Access (DMA) — des Systems verwendet. Der direkte Speicherzugriff erlaubt es dabei einem Ein/Ausgabegerät, die Daten direkt von oder zum Speicher zu transportieren. Der Prozessor selbst muß also die Daten hier nicht zur Verfügung stellen. Der Drucker beispielsweise benötigt den Prozessor, um jedes Zeichen zu senden, das gedruckt werden soll. Für den Datentransfer von und nach Disketten hätte der Prozessor allerdings Schwierigkeiten, die Daten schnell genug bereitzustellen. Ein Programm zum Übertragen von Daten von und nach Disketten wäre dann sehr ähnlich dem Programm in Abbildung 8.15, wo wir Zeichen an den Drucker sandten. Das heißt, das Programm würde in einer Schleife laufen und dabei jedesmal das RQM-Bit testen, um festzustellen, ob das nächste Datenbyte bereit ist. Antwortet in diesem Fall der Prozessor nicht schnell genug auf die Anforderungen des Diskettenlaufwerks, so sind jedoch die Daten verloren.

Beim Datentransport mit der DMA muß der Prozessor dagegen lediglich die Operation initialisieren. Der Intel 8237 DMA-Controller auf der Systemplatine erledigt dann den Rest für ihn. Für eine Diskettenleseoperation initialisiert das Programm beispielsweise die DMA so, daß sie den Datentransfer ausführen kann. Darauf sendet

unser Programm den Befehl an das Diskettenlaufwerk, die Leseoperation auszuführen. Während der eigentlichen Ausführungsphase muß unser Programm die Daten dann nicht mehr bewegen, denn der DMA-Controller erledigt diese Aufgabe. Ist die Operation ausgeführt, so führt unser Programm die Ergebnisphase wieder genauso durch wie vorher.

Sehen wir uns nun einmal an, wie wir die DMA für eine Diskettenleseoperation vorbereiten müssen. Im Abbildung 8.21 sehen wir die dazu nötigen Befehle. Die DMA verfügt über vier Kanäle. Das Diskettenlaufwerk ist dabei an den Kanal 2 der DMA angeschlossen. Die Kanäle 1 und 3 sind auf dem System-Ein/Ausgabekanal für andere Ein/Ausgabegeräte verfügbar, während der Kanal 0 für eine sehr wichtige Hardwarefunktion reserviert ist — das Auffrischen der Speicherinhalte. Stören wir also die Operation der DMA auf Kanal 0, so gehen wahrscheinlich große Teile des Speicherinhalts verloren.



```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 8.21  DMA Setup

1                                      PAGE    ,132
2                                      TITLE   Figure 8.21  DMA Setup
3
4         0000                 STACK   SEGMENT STACK
5         0000      40 [               DW      64 DUP(?)
6                        ????
7                             ]
8
9         0080                 STACK   ENDS
10
11        = 0000               DMA     EQU     0                ; I/O address of DMA chip
12        0000                 CODE    SEGMENT
13                                     ASSUME  CS:CODE
14        0000                 DMA_SET PROC    FAR
15        0000  1E                     PUSH    DS               ; Return address
16        0001  2B C0                  SUB     AX,AX
17        0003  50                     PUSH    AX
18
19        0004  B0 46                  MOV     AL,46H           ; Read from diskette
20        0006  E6 0B                  OUT     DMA+11,AL        ;  mode command to DMA
21        0008  E6 0C                  OUT     DMA+12,AL        ; Set first/last F/F
22        000A  8C C8                  MOV     AX,CS            ; Get current segment address
23        000C  B1 04                  MOV     CL,4
24        000E  D3 C0                  ROL     AX,CL            ; Rotate (multiply by 16)
25        0010  8A E8                  MOV     CH,AL            ; High four bits saved in CH
26        0012  24 F0                  AND     AL,0F0H          ; Zero out the low four bits
27        0014  05 0033 R              ADD     AX,OFFSET BUFFER ; Add in the offset of buffer
28        0017  73 02                  JNC     NO_HIGH_INCREMENT ; If there was a carry, then
29        0019  FE C5                  INC     CH               ;  the high four bits must increment
30        001B                 NO_HIGH_INCREMENT:
31        001B  E6 04                  OUT     DMA+4,AL         ; Output low byte of address
32        001D  8A C4                  MOV     AL,AH
33        001F  E6 04                  OUT     DMA+4,AL         ; Output high byte of address
34        0021  8A C5                  MOV     AL,CH
35        0023  E6 81                  OUT     081H,AL          ; Output to page register
36
37        0025  B8 01FF                MOV     AX,511           ; Count value for one sector
38        0028  E6 05                  OUT     DMA+5,AL         ; Low byte of count
39        002A  8A C4                  MOV     AL,AH
40        002C  E6 05                  OUT     DMA+5,AL         ; High byte of count
41        002E  B0 02                  MOV     AL,2             ; Enable channel 2
42        0030  E6 0A                  OUT     DMA+10,AL        ;  command to DMA
43        0032  CB                     RET
44        0033                 DMA_SET ENDP
45        0033  0200 [         BUFFER  DB      512 DUP(?)       ; Buffer for diskette read
46                     ??
47                        ]
48
49        0233                 CODE    ENDS
50                                     END
```

Abbildung 8.21 DMA-Vorbereitung

Jeder Kanal der DMA verfügt über zwei Register: ein Adressregister und ein Zählregister. Das Adressregister bestimmt dabei die Speicherstelle, an die die Daten transportiert werden sollen. Für unsere Leseoperation würde der Inhalt des Adressregisters beispielsweise den Anfang des Datenpuffers bestimmen. Während der Diskettencontroller nun die einzelnen Bytes von der Diskette liest, speichert der DMA-Controller diese Bytes an die vom Inhalt des Adressregisters bestimmten Speicherstellen. Der DMA-Controller erhöht daraufhin den Inhalt des Adressregisters, so daß dieses nun auf die nächste Position im Datenpuffer zeigt.

In Abbildung 8.21 wird dieser Datenbereich als BUFFER bezeichnet. Unser Programm bestimmt nun die absolute Adresse von BUFFER im System. Wir erreichen dies dadurch, indem wir auf den Adressoffset von BUFFER den bereits verschobenen Inhalt des CS-Registers addieren, das seinerseits die Segmentadresse von BUFFER enthält. Nun übergeben wir die niederwertigen 16 Bits dieser Adresse in das DMA-Adressregister für Kanal 2. Die höherwertigen vier Bits der Adresse werden in einem speziellen Seitenregister abgelegt. Der 8237 DMA-Controller kann nämlich in der Tat nur 16-Bit Adressen verarbeiten. Im IBM PC wird nun der Inhalt dieses Page-Registers auf den in der DMA enthaltenen Adresswert addiert, so daß unser Programm in der Tat jede beliebige Speicherstelle ansprechen kann. Es gibt übrigens drei Page-Register, je eines für Kanal 1, 2 und 3. Jedes Page-Register ist dabei nur vier Bits lang, so daß die vier höherwertigen Bits des AL-Registers keine Rolle spielen, wenn wir den Page-Wert ausgeben.

Unser Programm muß außerdem die Anzahl der zu übertragenden Bytes an den DMA-Controller übergeben. Auch der Diskettencontroller benötigt diesen Wert, den wir im Kanal 2 des Zählerregisters übergeben, um die Datenleseoperation korrekt zu beenden. Die DMA sendet dabei ein spezielles Kontrollsignal, das wir als „Terminal Count" bezeichnen, an das jeweilige Gerät, wenn sie das letzte Datenbyte im Speicher abgelegt hat. Der letzte Befehl an die DMA gibt dann Kanal 2 für weitere Operationen frei. Unser Programm kann nun in die Befehlsphase für den Diskettencontroller eintreten.

Der Diskettenadapter benötigt für seine Arbeit viele der einzelnen Systemkomponenten. So benötigt er zum Beispiel die DMA und einen zusätzlichen Interrupt, um die Arbeit des Diskettenlaufwerks zu überwachen. Der Diskettencontroller selbst ist ein kompliziertes „intelligentes" Steuergerät, das seinerseits ein Programm benötigt, bevor es unsere Befehle ausführen kann. Im nächsten Kapitel werden wir auf all dies näher eingehen, wenn wir besprechen, wie das ROM BIOS das Diskettenlaufwerk steuert.

9 ROM BIOS

Im vorhergehenden Kapitel beschäftigten wir uns mit der Hardware des IBM PC. IBM versieht den Rechner aber standardmäßig zusätzlich mit einem Steuerprogramm, das viele der besprochenen Hardwaregeräte bedient. Die Programme befinden sich im ROM und werden als ROM BIOS (Basic Input/Output-System) bezeichnet. Im folgenden Kapitel erklären wir nun genau die einzelnen Funktionen, die als Teil des ROM BIOS verfügbar sind. Sie sollten dieses Kapitel verwenden im Zusammenhang mit Kapitel 3 und Anhang A des Technical Reference Manual. Kapitel 3 beschreibt dabei das ROM BIOS, insbesondere einige seiner Spezialfunktionen. Und Anhang A enthält die komplette Assemblerliste des ROM BIOS, wie es im IBM PC eingesetzt wird.

Wir werden uns nun im weiteren aus zwei Gründen mit dem ROM BIOS befassen. Erstens sind diese Routinen ein gutes Beispiel für die Technik der Programmierung in Assemblersprache, besonders, was die Steuerung der Systemhardware betrifft. Zweitens, und noch wichtiger, ist die Tatsache, daß diese ROM BIOS-Routinen eine wichtige Rolle bei der Entwicklung von Assemblerprogrammen für den IBM PC spielen. Wir haben bereits in einigen früheren Beispielen bestimmte ROM BIOS-Funktionen verwendet. Da IBM uns diese Funktionen zur Verfügung stellt, gibt es keinen Grund, sie selbst zu schreiben. Es ist sogar eine sehr gute Idee, sie soweit als möglich zu verwenden.

Der zweite Grund zur Verwendung der Routinen des ROM BIOS ist die Portabilität von Programmen. IBM stellt die Routinen des ROM BIOS nämlich zur Verfügung, um dem Assemblerprogrammierer eine bestimmte Ebene von Systeminterface zu bieten. Als IBM den PC entwickelte, war den Systementwicklern sehr wohl klar, daß es in jedem Falle, so gut sie auch ihre Arbeit erledigten, immer einen Weg geben würde, noch etwas besser zu machen. Und in der Tat, mit fortschreitender Zeit wird auch die fortschreitende Entwicklung der Technik eine bessere Lösung für einige Aufgaben bieten.

Wird beispielsweise neue Hardware für den PC entwickelt, so werden auch neue Programmschnittstellen für diese Hardware zusätzlich nötig sein. Schreiben wir nun ein Programm, das sich auf ganz bestimmte Hardwarespezifikationen stützt, so werden wir dieses Programm jedesmal dann verändern müssen, wenn sich in der Hardware etwas verändert. Dies ist nun wahrlich kein großes Problem für einen Programmierer, der seine Programme selbst verwendet, kann aber zu einem Problem werden, wenn wir Programme schreiben, mit denen viele Leute umgehen. Oder sollten Sie beispielsweise Ihr Programm verkaufen wollen, so werden Sie sicher großes Interesse daran haben, daß nicht jede Hardwareänderung auch eine Änderung in Ihrem Programm erfordert. Das würde nämlich bedeuten, daß Sie nach jeder Hardwareänderung an Ihre Kunden ein neues Programm versenden müßten.

Die Interfaceroutinen des ROM BIOS sind ein Versuch, dieses Problem zu lösen. IBM hat dabei eine Schnittstelle definiert, mit der die verschiedenen Komponenten des Systems angesprochen werden können. Die Hardwarelieferanten ihrerseits werden nun dafür sorgen, daß neue Hardware soweit als möglich auf die bestehenden Schnittstellen paßt. Dies bedeutet, daß selbst bei einer Veränderung der Hardware

die Schnittstellen des BIOS die gleichen bleiben sollten. Ihr Programm muß also in diesem Fall nicht mehr verändert werden.

Verfügt natürlich die neue Hardware über zusätzliche Funktionen, dann müssen Sie Ihr Programm ändern, um auch einen Vorteil aus diesen neuen, zusätzlichen Funktionen zu ziehen. In diesem Falle wird IBM zusätzliche neue BIOS-Schnittstellen anbieten. Ihre alten Programme, die bisher perfekt liefen, werden selbstverständlich auch auf den neuen Maschinen laufen. Natürlich werden Sie in diesem Fall nicht von den neuen Funktionen profitieren. Erfüllte Ihr Programm bisher einen nützlichen Zweck, so ist es sehr wahrscheinlich, daß es seinen Wert dann immer noch besitzt und auch weiterhin verwendet wird.

Nehmen wir ein mögliches Beispiel aus dem ROM BIOS. Wir nehmen dazu an, daß IBM sich entschlossen hat, den Diskettencontroller zu verändern. Dies könnte aus verschiedenen Gründen geschehen. Beispielsweise könnte eine Verringerung der Herstellungskosten oder das Hinzufügen einiger zusätzlicher Funktionen ein Grund für die Modifikation des Diskettencontrollers sein. In allen Fällen wäre die Programmschnittstelle für die Hardware, so wie wir sie im vorausgehenden Kapitel besprochen haben, verschieden. Dagegen wird das ROM BIOS, das IBM zusammen mit dem Controller liefert, das gleiche Interface für uns besitzen. Dies bedeutet, daß der Aufruf und die Parameterübergabe für diese Schnittstelle gleich bleiben. Haben wir also ein Programm, das die Diskette benützt, und das für alle Diskettenzugriffe die Schnittstellen des ROM BIOS verwendet, so würde dieses Programm korrekt laufen. Hätten wir dagegen ein Programm geschrieben, das direkt auf den Diskettencontroller zugreift, so würde dieses Programm mit einer solchermaßen modifizierten Controllerkarte sicherlich nicht korrekt ablaufen.

Das ROM BIOS besteht aus mehreren, getrennten Funktionen. Die erste dieser Funktionen ist der Einschalt-Selbsttest (power-on self-test — POST). Diese Routine wird immer dann ausgeführt, wenn wir den Rechner einschalten oder wenn wir einen System-Reset (durch Drücken der Tasten Control-Alt-Del) durchführen. Die POST-Routine überprüft dabei die Hardware und initialisiert einige Geräte für nachfolgende Operationen.

Das ROM BIOS enthält zusätzlich Gerätetreiber. Diese Routinen dienen zur Steuerung der Geräte. IBM sah dabei Steuerungsroutinen für alle normalerweise verwendeten Hardwarezusätze vor. Allerdings steht dabei nicht jede denkbare bzw. für den Benutzer wünschenswerte Funktion zur Verfügung. Es gibt nämlich einfach nicht genug ROM-Speicherplatz, um alle eventuell wünschenswerten Funktionen aufzunehmen.

Schließlich verfügt das ROM BIOS noch über einen Satz von Systemdiensten. Diese Routinen steuern dabei nicht direkt irgendein Gerät, sondern versorgen uns mit wertvollen Daten über die Arbeit des Betriebssystems.

Im folgenden werden wir die einzelnen Komponenten des ROM BIOS besprechen. Wir beginnen dabei mit der POST-Routine, da sie einerseits die erste im ROM-Programm ist und andererseits auch die erste beim Einschalten des Rechners. Als nächstes folgen die Systemdienste, da sie in direktem Zusammenhang mit der Systemplatine stehen. Und schließlich werden wir uns noch ansehen, was IBM für das Arbeiten mit den verschiedenen Ein/Ausgabegeräten vorgesehen hat.

Hinweise zur ROM BIOS-Liste

Die Liste des ROM BIOS ist in Anhang A des Technical Reference Manual enthalten. Sie beschreibt den Inhalt des 8K-Speicherbereichs im ROM, der an den Adressen 0FE000H im Adressraum des 8088 beginnt. Dieser ROM ist einer von fünf Modulen, die sich auf der Systemplatine des IBM PC befinden. Die anderen vier ROM-Module enthalten den BASIC-Interpreter. In Analogie zum DOS ist dabei der Quellcode des BASIC-Interpreters Eigentum von IBM und deshalb nicht im Technical Reference Manual wiedergegeben. Dagegen hat IBM die ROM BIOS-Routinen veröffentlicht, so daß jedermann darin Einsicht nehmen kann, um sich die Schnittstellen der BIOS-Routinen anzueignen.

Die Programmliste in Anhang A des Technical Reference Manual enthält dabei den kompletten Inhalt des ROM. Die Assemblierung wurde nicht mit dem IBM Makro-Assembler durchgeführt. Dies geschah deshalb, da der Makro-Assembler zu der Zeit, als IBM das ROM BIOS entwickelte, noch nicht existierte. Aus diesem Grunde wurde für die Entwicklung des BIOS der Intel Makro-Assembler verwendet und erzeugte auch die gezeigte Liste. Der Intel-Assembler ist dabei dem IBM-Assembler in Anwendung und Syntax ähnlich. Wie wir allerdings sehen, gibt der Intel-Assembler die Adressbereiche nicht in der gleichen Art wieder, wie dies der IBM-Assembler tut; zusätzlich besteht ein Unterschied in einigen Assembler-Pseudo-Operationen. Es sollte für Sie jedoch ein leichtes sein, mit diesen geringfügigen Unterschieden fertig zu werden.

Das ROM BIOS beschreibt sechs verschiedene Segmente. Nur drei dieser Segmente sind wirklich von Bedeutung für uns. Das Segment ABS0, das an der Adresse 0 beginnt, enthält die Interrupt-Vektoren, mit denen das ROM BIOS und die POST-Routine arbeiten. Dieses Segment beinhaltet dabei keine vorher festgelegten Werte. Es bestimmt nur die Lage der Vektoren. Da diese sich im RAM befinden, muß das ROM BIOS nach Einschalten des Stroms diesen Bereich initialisieren. Das Datensegment, das sich bei Paragraph 40H bzw. an der absoluten Adresse 400H befindet, definiert nun alle vom ROM BIOS verwendeten Datenbereiche. In ähnlicher Weise bestimmt auch dieses Segment wiederum nur die Adressen der Variablen und nicht ihre Werte. Das Codesegment schließlich beginnt an der Paragraphenadresse 0F000H. Die ersten 56K Bytes dieses Segments sind leer. Das erste belegte Byte im Codesegment erscheint deshalb an der absoluten Adresse 0FE000H bzw. mit dem Offset 0E000H innerhalb des Segments. Diese Daten, von der Adresse 0FE000H bis zur Adresse 0FFFFFH, stellen nun die 8K Bytes des BIOS ROM-Moduls dar. Dieser Code, zusammen mit den BASIC ROM-Modulen, ist der einzige verfügbare Maschinencode, wenn die Maschine eingeschaltet wird.

Zusätzlich sei bemerkt, daß IBM das ROM BIOS nicht als einen einzigen großen Quellcode entwickelte. Jede Funktion wurde als eigener Modul entwickelt und die Module schließlich zusammengebunden, um das ROM BIOS zu bilden. Wir können noch einige Reste dieses Bindevorgangs im Code entdecken. Zur Veröffentlichung kombinierte IBM jedoch die einzelnen Codemodule zu einem einzigen großen Quellmodul und assemblierte diesen in einem Stück. Dies gestattet es uns nun, in der Assemblerliste die absoluten Adressen jeder einzelnen ROM BIOS-Funktion zu ermitteln.

Einschalt-Selbsttest

Der IBM PC führt nach jedem Rücksetzen des Systems, darin eingeschlossen auch das Einschalten des Rechners, einen Selbsttest durch. Dieser Test hat zwei Gründe: erstens ist er ein kurzer Test der grundlegenden Teile des Systems und zweitens initialisiert er die wichtigsten Hardwarekomponenten für die weitere Verarbeitung.

Hinsichtlich des Systemtests bildet die POST-Routine den ersten Teil eines dreiteiligen Diagnosepakets für den IBM PC. Die POST-Routine wird dabei ausgeführt, sooft das System eingeschaltet wird. Dieser kurze Test überprüft die Arbeit des Systems und entdeckt Probleme, bevor sie ein laufendes Programm betreffen können. Die zweite Diagnose-Ebene wird mit jedem Rechner im „Guide to Operations" mitgeliefert. Diese Diskette (oder Kassette) enthält ein Diagnoseprogramm für den Anwender, das jeden Teil des Rechners überprüft. Das Programm stellt dabei fest, welchen Teil des Rechners der Kunde an den Service einschicken sollte, falls ein Fehler auftaucht. Schließlich bietet IBM noch als drittes ein fortgeschrittenes Diagnosewerkzeug an. Dieses Programm, das nur gegen Aufpreis erhältlich ist, stellt fest, welche der austauschbaren Einheiten versagt hat. Dieses Hilfsmittel ist eigentlich zur Anwendung durch das Servicepersonal gedacht, wenn die Maschine gewartet wird. Es gibt allerdings keinen Grund, warum Sie dieses Programm nicht zu Ihren Zwekken kaufen könnten.

Der Anfang der POST-Routine ist etwas schwer zu finden. Dies rührt daher, daß sich der Anfang der POST-Routine im Grunde genommen am Ende des Programmlistings befindet. Wird der 8088 nämlich rückgesetzt (was zum Beispiel geschieht, wenn der Strom eingeschaltet wird), so beginnt die Abarbeitung an der Adresse 0FFFF:0000H. Diese Adresse befindet sich in einer Distanz von 16 Bytes vom Ende des Adressraums des 8088. In diesen 16 Bytes ist nun gerade noch genug Platz für einen Sprungbefehl auf die tatsächliche POST-Routine. Wie wir im weiteren sehen können, überträgt der Befehl FAR JMP die Steuerung dann an das Label START, wo sich der tatsächliche Anfang der POST-Routine befindet. Die restlichen verbleibenden Bytes am Ende des ROMs verwendet IBM, um einen Datenwert zu enthalten. Dieser Wert, ein Datum, teilt mit, wann IBM den ROM zur Produktion freigab.

Über den Anfang der POST-Routine können wir auch Rückschlüsse darauf ziehen, warum IBM den ROM für den PC an das obere Ende des Speicherbereichs plazierte. Dort beginnt der Prozessor nämlich nach einem Reset wieder mit dem Abarbeiten von Befehlen. Der System-ROM, der das Initialisierungsprogramm für den Rechner enthält, muß also einige Informationen an der Stelle 0FFFF:0000H enthalten. So ist es sinnvoll, gleich den ganzen ROM an das Ende des Speicherbereichs zu legen. Auch das Plazieren der Interruptvektoren an den Anfang des RAM-Bereichs erscheint nun sehr vernünftig. Die für uns dadurch entstehende Möglichkeit, den Inhalt dieser Vektoren zu verändern, erhöht die Anwendungsmöglichkeiten des ROM BIOS gewaltig.

Die POST-Routine enthält im übrigen für uns uninteressanten Code. Viele der Befehlsfolgen ergeben dabei überhaupt keinen Sinn. Wenn wir uns die erste Befehlsfolge einmal ansehen, so entdecken wir, daß darin überhaupt nichts geschieht – solange es kein Problem mit dem Prozessor selbst gibt. Solange wir

uns also nicht dafür interessieren, Diagnoseprogramme selbst zu erstellen, gibt es für uns wenig Grund, die Programmiertechniken der POST-Routine zu untersuchen.

Wir können allerdings einige der Aktionen festhalten, die POST ausführt, um Ihnen zu zeigen, in welchem Umfang Fehlertests durchgeführt werden. POST testet beispielsweise den gesamten ROM auf der Systemplatine mittels einer Prüfsumme.

Hardware-Unterbrechungen

Interrupt Nummer		Verwendung im ROM BIOS
2	02H	Parityfehler im Speicher
5	05H	Ausgabe auf Bildschirm
8	08H	Tageszeit
9	09H	Tastatur
14	0EH	Diskette

BIOS-Gerätetreiber

Interrupt Nummer		Verwendung im ROM BIOS
16	10H	Video
17	11H	Ausstattungstest
18	12H	Speichergröße
19	13H	Diskette
20	14H	Asynchron
21	15H	Kassette
22	16H	Tastatur
23	17H	Drucker
24	18H	Kassetten-BASIC
25	19H	Bootstraproutine
26	1AH	Tageszeit

Benutzerroutinen

Interrupt Nummer		Verwendung im ROM BIOS
27	1BH	Tastatur-Break
28	1CH	Zeitgeber-Tick

BIOS-Parameterblöcke

Interrupt Nummer		Verwendung im ROM BIOS
29	1DH	Videoparameter
30	1EH	Diskettenparameter
31	1FH	Video – graphischer Zeichensatz

Abbildung 9.1 Interruptvektoren für das ROM BIOS

Dabei werden alle Bytes des ROM-Moduls aufeinander addiert. Während dieser Addition werden sämtliche Überträge aus dem 8-Bit Ergebnis vernachlässigt. Ist das Endergebnis nun Null, so hat der ROM den Test bestanden. Natürlich hat IBM sichergestellt, daß jeder ROM eine Prüfsumme von Null ergibt, bevor er in den Rechner eingesetzt wird. Enthält der ROM dagegen eine schwache Stelle, so wird durch diesen Test der Fehler entdeckt.

Die POST-Routine testet auf diese Weise auch den gesamten Schreib/Lesespeicher des Systems. Die Schalter auf der Systemplatine teilen der POST-Routine dabei mit, wieviel Speicher sich im System befindet. Jedes Bit im Speicher wird getestet, um festzustellen, ob es auf 1 und dann auf 0 gesetzt werden kann. Nach Durchführung dieses Tests schreibt die POST-Routine an alle Speicherstellen Nullen. Das bedeutet, wenn wir ein Programm direkt im Anschluß an die POST-Routine laufen lassen, daß sich der gesamte Speicher auf 0 befindet. Es ist allerdings eine schlechte Gewohnheit, sich darauf zu verlassen, daß ein anderes Programm unseren Datenbereich initialisiert. Es ist besser und sicherer, dieses selbst durchzuführen.

Das letzte, das wir noch bei der POST-Routine festhalten wollen, ist, daß sie die Software-Interruptvektoren für das ROM BIOS initialisiert. Wir greifen nämlich aus einem normalen Programm über die Software-Interruptvektoren auf die Routinen des ROM BIOS zu. Die Routinen selbst befinden sich dabei in einem ROM-Modul, dem gleichen, der auch die POST-Routine enthält. Bevor nun POST die Steuerung an das Betriebssystem übergibt, stellt sie sicher, daß jeder der Einsprungpunkte in das BIOS im richtigen Interruptvektor abgespeichert ist. Das BIOS benützt dabei die meisten Interruptvektoren zwischen Interrupt 2 und Interrupt 01FH. Im Technical Reference Manual sehen wir eine Liste der Interruptvektoren, die uns die Interruptnummer und den Anfangswert des jeweiligen Vektors zeigt. Abbildung 9.1 enthält diesen Teil der Tabelle, den wir auch in unserer weiteren Besprechung des ROM BIOS verwenden werden.

Interrupts des ROM BIOS

Wie wir aus Abbildung 9.1 sehen, verwendet das ROM BIOS die Interruptvektoren des 8088. Diese Vektoren dienen verschiedenen Zwecken. Der erste Block von Vektoren behandelt direkt die Hardware-Interrupts. Die mit diesen Vektoren verbundenen Routinen erhalten dann die Steuerung, wenn ein Hardware-Interrupt auftritt. So verwendet beispielsweise die Tastatur den Interruptvektor 9, der sich an der Stelle 9*4 oder 24H befindet. Das ROM BIOS verwendet dabei nicht sämtliche Interrupts, die mit dem 8259 möglich wären. Einige dieser Interrupts sind für andere IBM-Geräte reserviert, während andere wiederum von uns für unsere eigenen Zwecke verwendet werden können. Und natürlich können wir, selbst wenn IBM einen solchen Interruptvektor für seine eigenen Zwecke reserviert hat, diesen für unsere Zwecke verwenden, wenn es in unserem System dafür vorgesehen ist. Wollen Sie allerdings Ihr Programm verkaufen, so sollten Sie sich darüber im Klaren sein, daß die Systeme von anderen Leuten dann nicht unbedingt mit Ihrem System übereinstimmen müssen.

Gerätetreiber

Die Gerätetreiber-Routinen sind das Herz des ROM BIOS. Diese Routinen geben dem Assemblerprogrammierer eine Möglichkeit zur Steuerung der Hardwareeinrichtungen des IBM PC. Jedes Programm kann dabei mit einer passenden Befehlsfolge die einzelnen Geräte ansprechen. Allerdings werden wir bei vielen Programmen gar nicht vorhaben, etwas besonderes mit den einzelnen Geräten zu tun. In diesen Fällen genügt es, daß die Geräte im vorbestimmten Standardmodus arbeiten. Beispielsweise tun nur wenige Programme mit dem Diskettenlaufwerk etwas anderes als lesen und schreiben. Wir haben beispielsweise unsere eigene Routine geschrieben, um den Diskettenlaufwerkstatus in Kapitel 8 zu erhalten. Wollen wir einen bestimmten Sektor auf der Diskette lesen, so können wir das ROM BIOS dazu verwenden und müssen nicht selbst erst das Programm zur Steuerung des Gerätes schreiben. Der Assemblerprogrammierer sollte also die Routinen des ROM BIOS als Werkzeug ansehen. Diese Routinen minimieren für ihn den Arbeitsaufwand.

Die Verwendung des ROM BIOS in unserem Programm geschieht durch die Ausgabe eines Software-Interrupts für die gewünschte Aktion. In den Registern des 8088 werden dabei die Parameter für und von den BIOS-Funktionen übergeben. Beispielsweise bestimmen die nachfolgenden Befehle den aktuellen Zustand des Bildschirms.

```
MOV     AH,15
INT     10H
```

Der Befehl INT 10H bewirkt den Aufruf der Video-BIOS-Routine. Die Videoroutine kann eine ganze Menge verschiedener Dinge für uns ausführen. Setzen wir das AH-Register dabei auf 15, wird der Routine mitgeteilt, daß wir etwas über den aktuellen Zustand des Bildschirms erfahren wollen. Die Routine selbst gibt uns dann die Statusinformation im AL-Register zurück.

Jeder der BIOS-Gerätetreiber verfügt über seine eigenen Parameter für Ein- und Ausgabe. Im allgemeinen dient dabei das AH-Register zur Identifizierung der bestimmten Funktion, die die BIOS-Routine ausführen soll. Die anderen Register werden von den BIOS-Routinen dazu verwendet, um zusätzlich benötigte Parameter für Ein-und Ausgabe zu übergeben. Das ROM BIOS-Listing enthält dabei die benötigten Spezifikationen für Ein- und Ausgabe jeder Routine. Am Anfang jeder BIOS-Routine sehen wir einen sogenannten Prolog, der die benötigten Eingabe- und Ausgabeparameter angibt. Jede Funktion ist außerdem kurz beschrieben und zusätzlich noch mit wichtigen Hinweisen versehen. Wir werden uns später noch auf diese Liste beziehen, wenn wir die einzelnen Routinen besprechen. Doch bevor wir dies tun, sehen wir uns zunächst noch einmal die anderen Interruptvektoren an.

Benutzerroutinen

Es gibt einige Systemfunktionen, die eine sofortige Beantwortung durch ein Programm erfordern. Zwei Interrupts für Benutzerroutinen sind dafür vorgesehen. Der erste Vektor, an der Interruptadresse 1BH, ist der „Keyboard Break". Der Benutzer

muß dabei nur die beiden Tasten „Control" und „Break" drücken, um das laufende Programm zu unterbrechen. Normalerweise wird dadurch die Steuerung an das Betriebssystem, wie z.B. DOS oder BASIC, zurückgegeben. Schreiben wir nun unser eigenes Programm, das diese Tastatur-Breakbedingung auswertet, so müssen wir dauernd die Tastatur überprüfen (beispielsweise über eine BIOS-Routine), um zu sehen, ob ein Benutzer gerade das Break-Zeichen gedrückt hat. Oder aber wir verwenden dazu den Break-Interrupt. Die ROM BIOS-Tastaturroutine gibt nämlich immer dann den Softwareinterrupt 1BH aus, wenn die Control-Break-Taste gedrückt wird. Normalerweise zeigt dieser Interrupt auf eine Rücksprungadresse — IRET — so daß nichts geschieht. Wollen wir aber, daß unser Programm etwas darüber erfährt, daß die Tasten Control-Break gedrückt wurden, so müssen wir nur den 1BH-Interruptvektor so besetzen, daß er auf die dazu vorgesehene Routine in unserem Programm zeigt. Auf diese Weise erfährt unser Programm sofort, wenn ein Benutzer es verlassen will, und kann dann die notwendigen Maßnahmen treffen.

Auf ähnliche Weise könnten wir ein Programm schreiben, das periodische Unterbrechungen benötigt. Beispielsweise müssen wir in einem Spielprogramm immer über die Position der jeweiligen Steuerhebel informiert sein. Das ROM BIOS ruft nun den Interrupt 1CH immer dann auf, wenn die Uhr im Rechner umschaltet. Wie wir bereits gesehen haben, geschieht dies 18,2 mal pro Sekunde oder etwa alle 55 Millisekunden. Wir können nun eine Routine schreiben, die die Position der Steuerknüppel jedes 18. Mal überprüft, wenn sie vom BIOS aufgerufen wird, und erlauben unserem Programm damit, die Position etwa einmal pro Sekunde festzustellen. Diese Methode gestattet uns einen periodischen Einsprung in eine bestimmte Routine innerhalb unseres Programms.

Parameterblöcke

Die Parameterblöcke des ROM BIOS geben den Hardwareprogrammen im ROM große Flexibilität. Die Interruptvektoren der einzelnen BIOS-Routinen zeigen dabei auf Parameterblöcke, die von den Routinen verwendet werden. So enthält beispielsweise der Diskettenparameterblock die Werte, die die BIOS-Routinen zur Steuerung des Diskettenlaufwerks verwenden. Da verschiedene Diskettenlaufwerke auch verschiedene Charakteristiken aufweisen, verfügt das ROM BIOS über jeweils eine Tabelle für die Geräte, die von IBM unterstützt werden. Haben Sie sich vielleicht entschlossen, ein anderes Diskettenlaufwerk zu verwenden, so müssen Sie nur diese Parametertabelle abändern und können dann sofort das Laufwerk erfolgreich verwenden.

Entsprechend gibt es auch eine Parametertabelle für die Initialisierung des Bildschirms. Benötigt Ihr Bildschirm beispielsweise einige geringfügig andere Zeitsignale, so können wir die dazu verwendete Tabelle modifizieren. Beispielsweise sind viele Fernsehbildschirme nicht in der Lage, die volle Breite von 40 Zeichen darzustellen. Einer der Parameter in der Bildschirmtabelle steuert dabei die Positionierung des linken bzw. rechten Randes auf dem Bildschirm. Das MODE-Kommando im DOS, mit dem wir auch die Zeichen auf dem Bildschirm verschieben können, modifiziert nun die Parametertabelle speziell zu diesem Zweck.

Der letzte Parameterblock, der mit BIOS-Interruptvektoren verbunden ist, ist eine Zeichentabelle. Das ROM BIOS verfügt nämlich über die Fähigkeit, Zeichen auf dem Bildschirm darzustellen, selbst wenn sich der Farb/Graphik-Bildschirm im Graphikmodus befindet. Dies geschieht durch Erzeugung der jeweiligen Zeichen aus dem dazu geeigneten Punktemuster. Wir können eine solche Tabelle im ROM sehen, die für die ersten 128 Zeichen des Zeichensatzes gilt und sich an der Stelle mit dem Offset 0FA6EH im Codesegment befindet. Der Interruptvektor 01FH zeigt auf die Tabelle mit den zweiten 128 Zeichen. Da im ROM nicht mehr genügend Platz für diese Tabelle war, ist es in den Händen des Benutzers, diese Tabelle zu liefern. Dies gestattet es uns, unseren eigenen Zeichensatz anstelle dessen zu verwenden, den IBM ursprünglich für die oberen 128 Werte gewählt hat. Wir müssen also nur unser eigenes Punktmuster entwickeln, den Interruptvektor 01FH so setzen, daß er auf die Tabelle zeigt und schließlich noch das BIOS im Graphikmodus schreiben lassen, um das gewünschte Zeichen auszugeben. Diese Fähigkeit kann eine sehr nützliche Erweiterung für Ihre Graphikprogramme sein. Sie gestattet es Ihnen nämlich, Ihre eigenen Zeichensätze zu entwerfen und auch zu verwenden.

Für alle Parameterblöcke gilt, daß wir nur den entsprechenden Interruptvektor modifizieren müssen, damit er auf die jeweiligen neuen Parameter zeigt. Wir können dabei die von uns verwendete Parametertabelle irgendwo in unserem Programm zur Verfügung stellen. Wir müssen nur den Interruptvektor so modifizieren, daß er dann auf diese Tabelle zeigt. Verwenden wir nun das ROM BIOS, und das BIOS benötigt an einer bestimmten Stelle einen Parameter, dann wird es in diesem Fall unsere Tabelle ansprechen und nicht die im ROM vorgesehene. Durch die Verwendung solcher Parametertabellen ist das BIOS extrem flexibel. Obwohl sich nämlich die eigentlichen Befehle im ROM-Modul befinden, können wir die Arbeit des ROM BIOS verändern, ohne jeweils einen neuen ROM einzusetzen oder die Routinen abändern zu müssen.

ROM BIOS-Datenbereich

Das DATA-Segment an der Paragraphenadresse 40H enthält die Variablen, die vom ROM BIOS verwendet werden. Wir werden uns jetzt nicht extra jede einzelne Variable und das, was sie tut, ansehen. Jede von ihnen ist nämlich in der entsprechenden Geräteroutine enthalten.

IBM wird die Adressen dieser Datenbereiche nicht ohne schwerwiegenden Grund ändern. Einige der ROM BIOS-Routinen tun nichts anderes, als einfach einen Wert aus diesen Datenbereichen zurückgeben. Es könnte für ein Programm deswegen durchaus vorteilhaft sein, diese Variablen direkt zu lesen. Wir werden im nächsten Kapitel ein Beispiel dafür bringen, das zusätzlich auch noch den Inhalt einer solchen Variablen verändern muß, die sonst vom ROM BIOS unterhalten wird. Ein Verändern dieser Werte gestattet uns eine zusätzliche Methode, das System zu benutzen.

Da IBM wahrscheinlich keine der Adressen dieser Variablen jemals verändern wird, ist es einigermaßen sicher, diese Datenbereiche direkt anzusprechen. Es gibt aller-

dings eine mögliche Einschränkung. Einige der Variablen könnten sich nämlich als unsinnig erweisen, wenn neue Hardwaregeräte entwickelt werden. Beispielsweise könnte IBM einen Rechner entwickeln, der über keinen Speicher verfügt (eine höchst unwahrscheinliche Möglichkeit), wo es dann keinen Sinn gäbe, eine Variable zu verwenden, die die Größe des aktuellen Speicherbereichs enthält. In diesem Falle könnte IBM auf den Gedanken kommen, die Variable für einen anderen Zweck zu verwenden. Doch solange es irgendeine Funktion geben wird, die der aktuellen Verwendung in etwa entspricht, so wird auch die verwendete Variable wahrscheinlich genauso weiter verwendet werden.

Gerätetreiberroutinen

Wir werden nun, eine nach der anderen, die einzelnen Gerätetreiberroutinen besprechen. Anstelle sie einfach der Reihe nach aufzuführen, werden wir sie uns in aufsteigender Schwierigkeit vornehmen. Die einfachsten Routinen sind die Systemdienste, also beginnen wir mit ihnen.

Systemdienste

Zwei ROM BIOS-Routinen führen auf verhältnismäßig einfache Art Systemdienste aus. Die beiden Routinen dienen zum Testen der Größe des Speichers und zum Überprüfen der angeschlossenen Geräte.

Die Routine zur Feststellung der Speichergröße verfügt über keine Parameter. Die Routine gibt ganz einfach im AX-Register die Größe des zur Verfügung stehenden Speichers zurück. Dabei enthält das AX-Register die Größe des verfügbaren Speichers in Tausenden von Bytes. Verfügt unser System also über 64K Bytes, so ist der im AX-Register zurückgegebene Wert 64. Jedes Anwendungsprogramm, das den Speicher bis zu einer bestimmten Größe verwendet, sollte diesen Wert zuerst lesen, um festzustellen, wo sich das Ende des aktuellen Speichers befindet. Ein Programm könnte selbstverständlich die Größe des zur Verfügung stehenden Speichers dadurch herausfinden, daß es Daten in den Speicher schreibt und diese dann wieder aus dem Speicher liest, solange, bis eine Speicherstelle gefunden wird, an der der gelesene Wert nicht mehr identisch ist mit dem, der zuerst geschrieben wurde. Allerdings ist es sinnvoller, wie ein Beispiel im nächsten Kapitel zeigen wird, in allen Programmen nur die Routine zum Ermitteln der Speichergröße zu verwenden, um das tatsächliche Ende des Speichers festzustellen. Es ist auf jeden Fall effizienter, weil wir nicht in jedem Programm eine eigene Routine schreiben müssen, die die Größe des Speichers feststellt. Durch ein Verändern der oberen Speichergrenze ist es außerdem möglich, einen bestimmten Teil im oberen Speicher zu reservieren. Nachdem nämlich ein Programm den Wert der Speichergröße verändert hat, wird ein anderes, ordentlich geschriebenes Anwendungsprogramm den Inhalt dieses Speicherbereichs nicht zerstören.

Auch die Routine zum Überprüfen der angeschlossenen Geräte benötigt keine Eingabeparameter. Auch sie gibt im AX-Register einen 16-Bit Wert zurück, der angibt, welche Geräte an dem jeweiligen System angeschlossen sind. Der Prolog für diese

Routine zeigt uns, welche Bedeutung dabei jedem einzelnen Bit zukommt. Diese BIOS-Funktion ist eine einfache Methode, um festzustellen, welches bestimmte Gerät vorhanden ist oder nicht.

Die letzte Systemdienstroutine behandelt schließlich noch die Tageszeit. Diese Routine hat zwei Funktionen: das Lesen der Zeit bzw. das Setzen der Zeit. Die Zeit wird dabei in Zeitgebersignalen seit Mitternacht gemessen oder dem Zeitpunkt, zu dem die Maschine eingeschaltet wurde. Die BIOS-Routine selbst wandelt diesen Wert dabei nicht in Stunden, Minuten oder Sekunden um. Aus der BIOS-Liste können wir jedoch die geeigneten Werte entnehmen, um diese Umsetzung durchzuführen. Um Stunden zu ermitteln, müssen wir nur den 24 Bit langen Zeitgeberwert durch 65.543 teilen, die Anzahl der Ticks pro Stunde. Um die Minuten zu ermitteln, genügt es, den Rest dieses Wertes durch 1092 zu dividieren, um die Anzahl der Ticks pro Minute zu erhalten usw.

Falls der solchermaßen erzeugte Zeitwert nicht besonders genau sein muß, gibt es auch einen einfacheren Weg zur Erzeugung. Da nämlich die Anzahl der Zeitgeberimpulse innerhalb 24 Stunden nicht in einem einzelnen Wort unterzubringen ist, ist der Zeitgeberwert eine drei Byte lange Integerzahl. Das höchstwertige Byte enthält dabei die Anzahl der Stunden mit einer Zeitabweichung von etwa einem Prozent. Das restliche Wort ergibt die Anzahl der Minuten, wenn man es durch 1092 dividiert. Eine Division des sich daraus ergebenden Rests durch 18 ergibt im weiteren die Anzahl der Sekunden.

Die Tageszeitfunktion benötigt einen Hardwareinterrupt, den Zeitgeber-Tick. Dieser Interrupt befindet sich auf Ebene 0 des 8259 und ist mit dem Softwareinterrupt 8 des 8088 verbunden. Die zugehörige Routine erhält alle 55 Millisekunden die Steuerung. Hauptaufgabe dieser Routine ist es, den Zeitgeber-Tick-Wert für die Tageszeit um jeweils 1 zu erhöhen. Es ist augenscheinlich, daß die Tageszeit nicht länger korrekt ist, wenn eine Programm diese Interruptsignale für eine bestimmte Zeit unterbricht.

Der Zeitgeber-Tick-Interrupt hat auch Bedeutung zur Steuerung des Diskettenlaufwerks. Der Motor des Diskettenlaufwerks läuft nämlich nicht dauernd. Dazu schaltet das BIOS den Motor nur für die Dauer des Diskettenzugriffs ein. Allerdings wird der Motor nicht direkt nach der Operation vom BIOS wieder ausgeschaltet. Es dauert nämlich einige Zeit, bis der Motor nach dem Einschalten genügend Geschwindigkeit erreicht hat, damit die Daten von der Diskette gelesen werden können. Macht nun ein Programm Diskettenzugriffe fast direkt aufeinander folgend, so ist es besser, den Motor laufen zu lassen, als ihn jeweils aus- und dann wieder einzuschalten. Die Zeitgeberroutine sorgt nun dafür. Die Diskettensteuerungsroutine ihrerseits setzt eine Variable, MOTOR_COUNT, jedesmal, wenn eine Diskettenoperation ausgeführt wurde. Der Zeitgeber-Tick-Interrupt erniedrigt nun diesen Wert. Hat die Variable MOTOR_COUNT den Wert 0 erreicht, so schaltet der Zeitgeber-Tick-Interrupt den Diskettenmotor aus. Die Diskettenroutine ihrerseits überprüft den Zustand des Motors, wenn sie einen Schreibbefehl auf der Diskette ausführen soll. Ist der Motor dabei bereits eingeschaltet, so gibt es keinen Grund, die Operation zu verzögern. Ist der Motor dagegen noch nicht eingeschaltet, so dauert es einige Zeit, bis die Diskette mit der korrekten Geschwindigkeit läuft. Normalerweise läuft der Dis-

kettenmotor noch etwa 2 Sekunden nach dem letzten ausgeführten Zugriff. Dies ist einer der Werte im Diskettenparameterblock, den Sie ohne weiteres verändern können. Die Wahl dieses Werts bestimmt das Verhältnis zwischen Performanz Ihres Programms und der Abnutzung der Diskette.

Alle drei besprochenen Serviceroutinen des BIOS übermitteln dabei nur Werte aus Speicherstellen an das aufrufende Programm. Wir können diese BIOS-Routinen umgehen, indem wir die entsprechenden Speicherstellen direkt lesen. Allerdings ist es in den meisten Fällen einfacher, die jeweilige BIOS-Routine aufzurufen, als die Adressierung über das BIOS-Datensegment herzustellen. Außerdem ist es von einem puristischen Standpunkt aus gesehen besser, die BIOS-Routinen zu verwenden.

Drucker und Asynchron-Schnittstelle

Die ROM BIOS-Routinen für den Drucker und für die asynchrone Schnittstelle ähneln sich sehr. Der hauptsächliche Unterschied besteht darin, daß die Asynchron-Adapterkarte die Fähigkeit besitzt, auch Zeichen zu lesen. Beide Routinen besitzen dabei Funktionen zum Initialisieren des Adapters, zum Senden von Zeichen und zum Lesen des aktuellen Zustands aus dem Adapter. In Abbildung 9.2 sehen wir die Funktionen, die für diese BIOS-Routinen verfügbar sind, zusammengefaßt.

ID (AH-Wert)	Druckerfunktion	Asynchronfunktion
0	Zeichen ausgeben	Adapter initialisieren
1	initialisieren	Zeichen senden
2	Status lesen	Zeichen empfangen
3	–	Status lesen

Abbildung 9.2 Funktionen von Drucker und Asynchronschnittstelle

Wie wir dabei aus der Tabelle sehen können, decken sich die beiden BIOS-Routinen nicht direkt. Der Wert im AH-Register nämlich, der die beiden Routinen ansteuert um bestimmte Funktionen auszuführen, ist jeweils verschieden. Aber das ist nun einmal so, und wir müssen damit leben.

Die BIOS-Routinen unterstützen jeweils mehr als eine Adapterkarte. Der BIOS-Datenbereich an der Adresse 40:0H verfügt über einen acht Worte langen Datenbereich. Dieser Bereich ist reserviert für die Basisadresse des Drucker- und des Asynchronadapters. Die vier Wörter beginnend bei Offset 0, bezeichnet als RS232_BASE, sind die Stelle, an der die Basisadressen von bis zu vier asynchronen Adapterkarten abgelegt werden können. Bei Offset 8, bezeichnet als PRINT_BASE, beginnt der entsprechende Datenbereich für die Druckeradapter. Die POST-Routine initialisiert diesen Datenbereich in Abhängigkeit von den Geräten, die sie im System vorfindet. Beim Drucker sieht die POST-Routine dabei zuerst auf der Schwarz/Weiß-Karte nach, dann an der Druckeradapteradresse 378H und schließlich an der Druckeradapteradresse 278H. Findet die POST-Routine einen Drucker-Adapter an einer dieser Adressen, so legt sie den Adresswert in dem entsprechenden Datenbereich

ab. Das gleiche geschieht mit den Adressen der Asynchronadapter, wobei zuerst bei der Adapterkarte mit der Ein/Ausgabeadresse 3F8H und dann bei der Karte mit der Adresse 2F8H nachgesehen wird.

Die BIOS-Routinen sind also unabhängig von den jeweiligen Ein/Ausgabeadressen der entsprechenden Adapterkarte. Im DX-Register müssen wir deswegen einen Inputparameter zur Verfügung stellen, der angibt, welche der verfügbaren Karten vom BIOS zu verwenden ist. Haben wir beispielsweise eine Schwarz/Weiß-Karte mit der Drucker-Ein/Ausgabeadresse 3BCH, so erscheint diese Adresse als erste in der Tabelle PRINT_BASE. Rufen wir nun die BIOS-Druckroutine auf und setzen dabei den Wert des DX-Registers auf 0, so werden vom BIOS alle Ein/Ausgabe-Operationen über diese Karte ausgeführt. Haben wir dabei zusätzlich noch einen weiteren Druckeradapter, der sich an der Ein/Ausgabeadresse 378H befindet, so bewirkt das Setzen des DX-Registers auf den Wert 1, daß alle BIOS-Ein/Ausgabebefehle sich nun mit dieser Karte beschäftigen. Wenn wir uns den Code für den Drucker und den Asynchrontreiber im BIOS ansehen, so stellen wir fest, daß der Wert des DX-Registers dazu verwendet wird, um die entsprechende Adresse in der Basisadresstabelle im BIOS-Datensegment zu ermitteln. Nachdem vom BIOS dieser Adresswert ermittelt wurde, werden alle weiteren Ein/Ausgabebefehle mit dieser Adresse, mit einigen Modifikationen, ausgeführt. Die Routinen enthalten außerdem eine Anzahl von Increment- und Decrementbefehlen für das DX-Register. Dies erlaubt es der BIOS-Routine, die verschiedenen Register der Ein/Ausgabekarten zu adressieren, ohne absolute Werte zu verwenden. Alle Ein/Ausgabereferenzen sind also relativ zu den Originalwerten, die aus der Basisadresstabelle ermittelt wurden.

Die Druckerinitialisierungsfunktion benötigt keine Inputparameter vom Benutzer. Das Initialisierungsprogramm setzt dabei den Drucker zurück und bereitet den Steuerport des Druckers für weitere Aktionen vor. Die Initialisierung der RS232-Schnittstelle auf der anderen Hand benötigt vom Programmierer Informationen über die Leitungsparameter. Einzelheiten über diese Initialisierungswerte, wie sie im AL-Register übergeben werden, sind im Prolog der Asynchronroutine enthalten.

Die anderen BIOS-Funktionen für Drucker und asynchrone Schnittstelle erlauben es uns, Daten auf das jeweilige Gerät zu schreiben (im Falle der Asynchronschnittstelle auch zu lesen). Das wichtige an diesen BIOS-Routinen ist, daß die Ein/Ausgabeoperationen synchron ablaufen. Dies bedeutet, daß ein Programm, nachdem es die Steuerung an die BIOS-Routinen übergeben hat, um die gewünschte Funktion auszuführen, die Steuerung erst dann wieder erhält, wenn diese Funktion vollkommen ausgeführt ist. Wenn wir ein Zeichen auf den Drucker ausgeben, behält also die Druckerroutine so lange die Steuerung, bis dieses Zeichen an den Drucker übermittelt ist. Ist der Drucker dabei noch nicht verfügbar, so läuft die BIOS-Routine in einer Schleife, um so darauf zu warten, bis der Drucker seine aktuelle Tätigkeit beendet hat. Veranlassen wir das BIOS, über die asynchrone Schnittstelle ein Zeichen zu senden, so wartet die Routine solange, bis die Hardware das Senden eines weiteren Zeichens zuläßt. In ähnlicher Weise wartet die Empfangsroutine der asynchronen Schnittstelle solange, bis der Adapter ein Zeichen empfangen hat. Sendet das gewünschte externe Gerät niemals ein Zeichen, so erhält das Programm, das die entsprechende BIOS-Routine aufgerufen hat, niemals wieder die Steuerung.

Aus diesem Grund verfügen beide Routinen über eine Statusfunktion. Diese erlaubt es unserem Programm, festzustellen, ob die BIOS-Routine zur aktuellen Zeit die entsprechende Operation ausführen kann. Die Druckerstatusfunktion teilt uns dabei mit, ob der Drucker augenblicklich beschäftigt ist. Die Asynchron-Statusroutine teilt uns mit, ob ein Zeichen übermittelt werden kann bzw. ob ein Zeichen gelesen werden kann. In einem Programm können wir diese Statusinformation auswerten, um festzustellen, ob wir sofort im Ablauf weiterfahren können. Wir könnten uns nämlich anderenfalls entscheiden, etwas anderes in unserem Programm auszuführen, bis die benötigte Ein/Ausgabeoperation verfügbar ist. Testen wir beispielsweise, ob irgendein äußeres Ereignis, wie der Empfang eines Zeichens durch die Karte, eintritt, so hält die Statusroutine unser Programm solange nicht auf, bis das Zeichen nicht tatsächlich gesendet ist. Dieser Test, ob das Zeichen wirklich angekommen ist, erlaubt es uns in der Zwischenzeit, fortzufahren andere Dinge zu tun.

Ein weiterer wichtiger Gesichtspunkt der BIOS-Routinen ist ihre Art der Fehlerbehandlung. Unter Verwendung der BIOS-Routinen ist es nämlich extrem schwierig, das System „aufzuhängen". Mit Ausnahme des Falls, in dem wir auf ein Zeichen von einem externen, asynchronen Gerät warten, kehren die BIOS-Routinen immer wieder zum aufrufenden Programm zurück, selbst wenn das externe Gerät Fehler liefert. Dazu wird vom BIOS ein Zähler verwendet, um auf die Vollendung der Aktion zu warten, die von der Hardware ausgeführt werden soll. Wartet beispielsweise die Druckerroutine darauf, daß der Drucker seine laufende Tätigkeit beendet, so enthalten das BL- und das CX-Register einen Zählwert. Erreicht der Wert in diesen Registern Null, bevor der Drucker wieder verfügbar ist, so kehrt die BIOS-Routine mit einem Time-out-Fehler an das aufrufende Programm zurück. Dies bedeutet, daß z.B. ein Abschalten des Druckers während wir versuchen, ein Zeichen auf diesen auszugeben, nicht dazu führt, daß das System stehenbleibt. Allerdings kehrt die BIOS-Routine in diesem Fall von der Befehlsausführung mit einer Fehlermeldung zurück, die anzeigt, daß beim Drucker ein Fehler auftrat.

Bei der Druckerroutine ergibt sich zusätzlich mit diesen Time-out-Werten noch ein geringfügiges Problem. Senden wir nämlich das Zeichen für Formularvorschub (0CH) an den Drucker, so schiebt der Drucker das Papier bis zum Anfang der nächsten Seite vor. Sind dabei noch mehr als 51 Zeichen in der gerade laufenden Seite, so dauert der Papiervorschub auf dem Drucker länger, als der Time-out-Wert dies erlaubt. Es ist also in diesem Falle möglich, eine Fehlerbedingung zu erhalten, selbst wenn der Drucker korrekt arbeitet. Der entsprechende Wert wurde deshalb in der zweiten Version des ROM BIOS verändert, um das Problem zu lösen. Verfügt Ihr System allerdings noch über die Originalversion des BIOS, so kann es notwendig sein, eine Druckeroperation, die mit einem Time-out-Fehler beendet wird, zu wiederholen. Dieses Wiederholen der Funktion stellt sicher, daß es sich nicht um einen BIOS-Fehler handelt.

Tastatur

Die BIOS-Tastaturroutinen bieten ein sehr gutes Beispiel für verschiedene Anwendungs- und Programmiertechniken. Erstens verfügt die BIOS-Tastaturroutine über einen großen Programmteil zur Unterbrechungsbehandlung, der die Konversion der Zeichen von Tastatur-Scancodes in den eigentlichen ASCII-Code bewirkt. Die Interruptroutine behandelt dabei die normalen Großschreibungstasten und die aus- und einschaltbaren Großschreibungstasten. Außerdem werden bis zu 15 Tastaturanschläge in einem Ringpuffer gespeichert, was es uns gestattet, schneller zu schreiben, als ein Programm in der Lage ist, die einzelnen Tastaturanschläge zu akzeptieren.

Tastaturdaten

Der Tastatur-BIOS-Datenbereich beginnt mit dem Adressoffset 17H im Datensegment. Die beiden Flagvariablen, KB_FLAG und KB_FLAG_1 sind bitsignifikant und lassen uns die Stellung der einzelnen Großschreibungstasten festlegen. Die Gleichsetzung nach der Definition der einzelnen Variablen zeigt die Bedeutung der einzelnen Bits. Beispielsweise enthält das Bit 3 der Variablen KB_FLAG den Zustand der Alt-Taste. Ist sie gedrückt, so steht dieses Bit auf „1". Ist die Taste nicht gedrückt, so steht dieses Bit auf „0". Die Bits der Variablen KB_FLAG spiegeln dabei den aktuellen Zustand aller Tastatur-Großschreibungstasten, also sowohl der normalen, wie auch der feststellbaren. Die feststellbaren Großschreibungstasten verwenden nun außerdem die Bits der Variablen KB_FLAG_1. Die Tasten verändern den Zustand der Tastatur jedesmals, wenn sie gedrückt werden. So schaltet beispielsweise die Taste CAPS LOCK die Tastatur von Großschreibung auf Kleinschreibung bzw. umgekehrt, immer wenn sie gedrückt wird.

Das BIOS verwendet die Bits in Flag 1, um festzustellen, ob die CAPS LOCK-Taste (oder eine andere Umschalttaste) aktuell gedrückt ist. Dies ist notwendig, da alle Tasten der Tastatur mit einer Wiederholfunktion ausgestattet sind. Würde nämlich das BIOS die Variable CAPS_STATE jedesmal umschalten, wenn ein Arbeits-Scancode von der Taste CAPS LOCK ankommt, so würde es die Wiederholfunktion der Taste für den Benutzer der Tastatur unmöglich machen, festzustellen, in welchem Status die Tastatur sich gerade befindet. Das BIOS schaltet das Bit in CAPS_STATE deshalb immer dann um, wenn der erste Arbeits-Scancode ankommt. Alle weiteren Arbeits-Scancodes werden ignoriert, bis der erste Ruhe-Scancode von der Taste CAPS LOCK eintrifft, der anzeigt, daß die Taste wieder losgelassen wurde.

Die Variable ALT_INPUT wird vom BIOS dazu verwendet, um die speziellen Zeichen festzuhalten, die eingegeben werden, wenn die Taste ALT zusammen mit der Shift-Taste gedrückt wird. Ist diese Taste nämlich gedrückt, so können wir einen Dezimalwert über das numerische Tastenfeld eingeben. Die Tastaturroutine gibt den der Dezimalzahl entsprechenden ASCII-Zeichenwert zurück, wenn wir die ALT-Shift-Taste loslassen. Diese Technik gestattet uns, jeden beliebigen Zeichencode auf dem IBM PC einzugeben, selbst wenn dieser nicht auf der Tastatur darstellbar ist. Wir drücken nun beispielsweise die Taste ALT, die Großschreibungstaste, und tippen dann auf der numerischen Tastatur die Werte „1", „1", „1" ein und lassen nun die Tasten wieder los. Als Folge erscheint das Zeichen „o" auf dem Bildschirm. Dieses Zeichen hat den Dezimalwert 111 im ASCII-Zeichensatz.

Die Variable ALT_INPUT enthält dabei den laufenden ASCII-Wert, der eingegeben wurde, während die Tasten ALT und Shift gedrückt sind. Wird nun über die Zifferntastatur eine Zahl eingegeben und ist der Tastaturzustand ALT-Shift, so wird vom BIOS der in ALT_INPUT enthaltene Wert mit 10 multipliziert und der neue Wert darauf addiert. Werden die Tasten ALT und Shift losgelassen, so wird der in der Variablen ALT_INPUT enthaltene Wert als Zeichen ausgegeben. Der Inhalt der Variablen ALT_INPUT ist normalerweise 0, was vom BIOS nicht als gültiger Wert für die Kombination der Tasten ALT, Shift und der Zifferntastatur betrachtet wird. Dies gestattet es dem Benutzer, die ALT- und Shift-Tasten auch in Verbindung mit anderen Tasten zu verwenden, beispielsweise in BASIC, wo die Tastenkombination ALT und A den Zeichenstring AUTO erzeugt, ohne daß vorher jedesmal 0 eingegeben werden muß, wenn die Taste ALT in Zusammenhang mit Shift und der numerischen Tastatur verwendet werden soll.

Die restlichen Tastaturvariablen bilden den Tastaturpuffer. Werden dabei Tasten auf der Tastatur angeschlagen, so wird ein Interrupt erzeugt. Die BIOS-Routine KB_INT übernimmt diesen Tastaturinterrupt, liest den Scancode von Port 60H und bestimmt den ASCII-Wert der gedrückten Taste. Der Wert wird dann vom BIOS in den Tastaturpuffer der Variablen KB_BUFFER gespeichert. Dieser Puffer hat eine Gesamtlänge von 16 Wörtern — jeder Tastendruck wird als Wortwert gespeichert. Das erste Byte entspricht dabei dem ASCII-Wert der gedrückten Taste, während das zweite Byte den Scancode bzw. den erweiterten Scancode der gedrückten Taste enthält. Die Verwendung des erweiterten Scancodes gestattet es uns, auch Tastenwerte an das Benutzerprogramm weiterzugeben, die nicht mit einem ASCII-Wert versehen sind.

Für den Tastaturpuffer existieren zwei Pointer. Die Variable BUFFER_HEAD enthält dabei den Offset im Datensegment für das erste Zeichen im Puffer, also den Wert der Taste, deren Betätigung am weitesten zurückliegt. Die Variable BUFFER_TAIL zeigt dagegen auf den Wert der Taste, der als letzter eingegeben wurde. Haben beide Pointer den gleichen Wert, so ist der Puffer leer.

Erhält die BIOS-Interruptroutine nun einen gültigen Tastaturcode, so wird dieser Wert im Puffer abgelegt. Ist der Puffer dabei noch nicht voll, so wird von der Tastaturinterruptroutine das Zeichen an der Stelle abgelegt, die durch den Wert der Variablen BUFFER_TAIL bestimmt ist. Danach wird die Variable um 2 erhöht, damit sie auf die nächste freie Stelle im Puffer zeigt. Würde das Erhöhen des Pointers dabei dazu führen, daß er an eine Stelle außerhalb des Puffers zeigt, so wird der Pointer wieder auf den Anfang des Puffers gesetzt. Dies bedeutet also, daß der Tastaturpuffer umklappt. Nach jedem jeweils sechzehnten Tastendruck wird das nächste Zeichen wieder am Anfang des Puffers abgelegt. Diese Anordnung wird auch als Ringpuffer bezeichnet, da die einzelnen Positionen innerhalb des Puffers so behandelt werden, wie wenn der Puffer als Kreis angeordnet wäre.

Die BIOS-Tastaturroutine für Interrupt 16H hat drei Aufgaben. Eine Funktion entnimmt ein Zeichen aus dem Puffer. Die Variable BUFFER_HEAD zeigt dabei auf das erste Zeichen im Puffer. Falls der Puffer nun nicht leer (angezeigt dadurch, daß die Variable „Head" gleich der Variablen „Tail" ist) ist, nimmt das BIOS das Wort an der Stelle BUFFER_HEAD und erhöht dann auch diesen Pointer um 2. Auch hier wird der Pointer wieder auf den Anfang des Puffers gesetzt, wenn sein Wert den in

BUFFER_ END enthaltenen Wert überschreitet. Das Unterprogramm K4 im Tastatur-BIOS führt dieses Erhöhen des Pointers und nötigenfalls das Umklappen durch.

Eine weitere Tastaturfunktion gibt den aktuellen Status des Tastaturpuffers zurück. Dadurch wird dem aufrufenden Programm mitgeteilt, ob sich im Tastaturpuffer ein Zeichen befindet oder nicht. Ein Programm könnte diese Information beispielsweise dazu verwenden, um Wartezeiten auf einen Tastaturanschlag zu verhindern, solange es in der Zwischenzeit auch noch andere Operationen ausführen könnte. Wir könnten solch einen Statusaufruf verwenden, um festzustellen, wann wir eine bestimmte Schleife zu verlassen haben. Bei jedem Durchlauf durch die Schleife können wir solchermaßen testen, um festzustellen, ob bereits ein Zeichen auf der Tastatur eingegeben wurde. Falls nicht, fahren wir einfach in unserer Schleife weiter. Wurde dagegen eine Taste gedrückt, so wird die Schleife beendet. Würden wir dagegen an dieser Stelle die BIOS-Funktion verwenden, die nur den Tastaturwert liest, so würde unsere Schleife niemals ausgeführt.

In Abbildung 9.3 sehen wir eine Assemblerroutine, die jeweils 1 auf eine vier Byte lange Integerzahl addiert, wenn die Schleife durchlaufen wird. Drückt der Benutzer dagegen die Leertaste, so wird die Routine verlassen. Die Routine verwendet dazu zwei der Tastatur-BIOS-Funktionen. Ist der Wert im AH-Register auf 1 gesetzt, so gibt uns das BIOS den Status des Tastaturpuffers zurück. Ist dabei das Nullflag gesetzt, so ist kein Zeichen vorhanden. Ist dagegen ein Zeichen vorhanden, so muß unsere Routine außerdem dieses Zeichen lesen. Sonst bleibt nämlich dieses Zeichen im Puffer stehen, bis das nächste Programm (oder in unserem Beispiel das gleiche Programm, aber später in seiner Ausführunsphase) wiederum nach einem Zeichen fragt. Im Beispiel wird das Zeichen aus dem Tastaturpuffer entnommen, in dem die gleiche BIOS-Funktion verwendet wird, wobei der AH-Wert auf 0 gesetzt ist. Das BIOS gibt dann im AL-Register das Zeichen zurück und unsere Routine vergleicht diesen Wert mit dem Wert eines Blanks. Wir sehen also außerdem, wie wir solchermaßen bei jedem Schleifendurchlauf auf ein bestimmtes Zeichen testen können.

Innerhalb des Tastatur-BIOS

Wir wollen das Tastatur-BIOS nun nicht Zeile für Zeile durchgehen. Es gibt jedoch etliche Programmabschnitte, die von Interesse sind. Einige von diesen haben wir bereits erwähnt, so die Routine K4, die dazu verwendet wird, die Pufferzeiger zu erhöhen.

Die Routine KB_INT verwendet verschiedene Tabellen mit Werten für Tastaturcodes. Gehen wir den Code durch, so sehen wir, daß diese Tabellen auf verschiedene Art verwendet werden. Die Tabellen mit Scan-Codes werden beispielsweise zum Testen auf gleiche Bitmuster verwendet. Das BIOS vergleicht die Scancodes, die von der Tastatur kommen, mit den Werten in dieser Tabelle. Mit dem Befehl REPNE SCASB, wie wir ihn direkt nach dem Label K16 sehen, sieht das BIOS in der Tabelle nach, ob dieser Wert mit einem der Werte der Großschreibungstasten übereinstimmt. Wird in der Scancodetabelle eine Übereinstimmung entdeckt, so wird der Offset in dieser Tabelle dazu verwendet, den Bitmaskenwert für die Variable

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 9.3   Keyboard Status

1                                          PAGE    ,132
2                                          TITLE   Figure 9.3   Keyboard Status
3       0000                       STACK   SEGMENT STACK
4       0000       40 [                    DW      64 DUP(?)
5                        ????
6                              ]
7
8       0080                       STACK   ENDS
9       0000                       CODE    SEGMENT
10                                         ASSUME  CS:CODE
11      0000  0000                 LITTLE  DW      0
12      0002  0000                 BIG     DW      0
13      0004                       COUNT   PROC    FAR
14      0004  1E                           PUSH    DS
15      0005  2B C0                        SUB     AX,AX            ; Set return address
16      0007  50                           PUSH    AX
17      0008                       ADD_ONE:
18      0008  2E: FF 06 0000 R             INC     LITTLE
19      000D  75 05                        JNZ     STILL_LOW
20      000F  2E: FF 06 0002 R             INC     BIG
21      0014                       STILL_LOW:
22      0014  B4 01                        MOV     AH,1             ; Keyboard status routine
23      0016  CD 16                        INT     16H
24      0018  74 EE                        JZ      ADD_ONE          ; No character available
25      001A  B4 00                        MOV     AH,0
26      001C  CD 16                        INT     16H              ; Read the character
27      001E  3C 20                        CMP     AL,' '           ; Is it a space
28      0020  75 E6                        JNZ     ADD_ONE
29      0022  CB                           RET                      ; All done
30      0023                       COUNT   ENDP
31      0023                       CODE    ENDS
32                                         END     COUNT
```

Abbildung 9.3 Tastaturstatus

KB_FLAG zu ermitteln. Da alle Shift-Tasten als Bits in den Flagvariablen repräsentiert sind, kann somit eine einzelne Routine, zusammen mit diesen Tabellen, alle Shift-Tasten kontrollieren.

Das BIOS verwendet noch weitere Tabellen, um beispielsweise die Scancodes in ASCII-Codes umzuwandeln. Ist der aktuelle Shift-Status einmal ermittelt, so verwendet das BIOS das BX-Register als Zeiger auf die aktuelle ASCII-Tabelle. Der Tastatur-Scancode wird dann in den korrekten Anfangswert für die gewünschte Tabelle umgesetzt (durch Subtraktion des Anfangswerts der Tabelle). Der Befehl XLAT ergibt im folgenden den korrekten ASCII-Wert für den ursprünglichen Scancode. Diese Methode wird beispielsweise bei Label K63 verwendet, wo das BIOS Pseudo-Scancodes für die numerische Tastatur bei Verwendung der gedrückten Control- und Shift-Taste erzeugt.

Das Unterprogramm ERROR_BEEP ist außerdem noch ein Beispiel für die Steuerung des Lautsprechers, wie wir es bereits in einem vorhergehenden Kapitel erläutert haben. Das BIOS erzeugt diesen Ton, sobald der Benutzer ein Zeichen eingibt und der Tastaturpuffer voll ist. Da dieser Ton immer dann ertönt, wenn das System gerade eine Tastaturunterbrechung abhandelt, wäre es unklug, den Wert des Zeitgeberkanals zu verändern, der den Lautsprecher betreibt. Aus diesem Grund verwendet das BIOS hier die direkte Steuerung des Lautsprechers. Wird also zu dieser Zeit irgendein anderer Ton erzeugt, so wird dieser unterbrochen und der Pufferüberlaufton ertönt. Wenn Sie dabei genau auf diesen Überlaufton hören, werden Sie feststellen, daß er leicht zittert. Der Zeitgeberinterrupt, der 16 mal pro Sekunde auftritt, unterbricht nämlich die Zeitschleife und modifiziert damit konsequenterweise den ausgegebenen Ton. Wie bereits im vorangegangenen Kapitel vorgeschlagen, können Sie hier die Folgen der Verwendung verschiedener Zeitschleifen für die Ausgabe von Tönen auf dem Lautsprecher untersuchen. Ein passendes Beispiel haben Sie ja.

Kassette

Das Kassetten-BIOS ist ein Beispiel für die Steuerung eines seriellen Geräts mit Zeitschleifen. Allerdings verwendet das Kassetten-BIOS wegen der verschiedenen Zeitbedingungen der einzelnen Befehle den 8253 Zeitgeber/Zähler für alle zeitlich kritischen Aufgaben. Wir sehen uns einmal zwei der Zeitroutinen, READ_HALF_BIT und WRITE_BIT an.

Das Technical Reference Manual enthält dazu eine komplette Beschreibung der Verschlüsselungsart für Daten, die auf der Kassette gespeichert werden. Die Routine WRITE_BIT schreibt dabei ein einzelnes Datenbit auf das Band. Der Ausgang von Kanal 2 des Zeitgebers/Zählers ist direkt mit dem Kassettenausgangsport verbunden. Das Schreiben eines Datenbits besteht also darin, die korrekte Frequenz in den Zähler des Kanals 2 zu setzen und dann nur noch zu warten, bis ein Zyklus vollendet ist. Die Routine WRITE_BIT erfüllt diese Aufgabe, allerdings in der umgekehrten Richtung. Wenn WRITE_BIT nämlich die Steuerung übernimmt, dann wird das vorhergehende Bit gerade noch geschrieben. Die beiden Zeit- bzw. Warteschleifen in WRITE_ Bit warten deshalb bis zur vollständigen Ausführung eines jeden Halbzyklus für das vorhergehende Bit. Ist das Bit fertig ausgegeben, so plaziert das BIOS die neue Frequenz in den Kanal 2 des Zeitgebers. Die Routine WRITE_BIT kehrt dann bereits zum aufrufenden Programm zurück, während die neue Frequenz gerade auf das Magnetband ausgegeben wird. Die Kassettenroutine ist also schnell genug (oder die Übertragungsrate auf die Kassette ist langsam genug — es hängt vom jeweiligen Standpunkt ab), damit die Routine WRITE_BIT wieder aufgerufen werden kann, bevor der Zeitgeber den ersten Halbzyklus für das jeweilige Bit beendet hat.

Die Routine READ_HALF_BIT führt die genau entgegengesetzte Arbeit aus. Sie wartet nämlich, bis das Kassetteninputbit (Bit 4 des Ports 62H) den Status ändert. Jede Änderung dieses Bits entspricht dabei der Hälfte eines auf dem Magnetband aufgezeichneten Bits. Das Kassetten-BIOS subtrahiert dann den aktuellen Zeitgeberwert vom Wert des Zeitgebers beim vorhergehenden Bitwechsel. Der entstehende Wert gibt die Zeit wieder, die der Kassetteneingang benötigte, um von einem Status in den anderen zu wechseln. Die Addition der beiden Halbbitdurchgänge erzeugt dann die vollständige Länge eines Zyklus für 1 Bit. Da diese Zykluszeiten unterschiedlich bei Einsen und Nullen sind, kann die READ_BYTE-Routine den Wert des jeweiligen Bits feststellen. Acht dieser solchermaßen gelesenen Bits werden dann kombiniert und ergeben ein einzelnes Byte.

Die Routine READ_HALF_BIT zeigt außerdem die Verwendung des Zeitgeberkanals 0 für Zeitmeßzwecke. Dazu wird der im Zeitgeber enthaltene Wert eingefroren und in das AX-Register übertragen. Die Verwendung des Werts 0 im Zähler des Zeitgebers 0 erlaubt es uns dabei, alle beliebigen Zeitgeberwerte voneinander abzuziehen, ohne Rücksicht darauf nehmen zu müssen, welcher größer oder kleiner ist. In allen Fällen wird sich die korrekte Differenz ergeben.

Die Kassetten-BIOS-Routine verfügt über vier Funktionen. Zwei von diesen sind Block-Ein/Ausgabe-Operationen — lies einen Datenblock bzw. schreibe einen Datenblock. Aus Gründen der Bandersparnis sind die Daten auf der Kassette in

Blöcken von je 256 Bytes gespeichert. Das BIOS überprüft diese Blöcke auf Fehler mittels einer zyklischen Redundanzprüfung (cyclic redundancy check — CRC). Mittels diese CRC-Prüfung ist es möglich, beinahe jeden Fehler aufzudecken, der mit dem Magnetband geschehen kann. Dies gestattet es uns, auf dem IBM PC das Kassettengerät als Speichermedium zu verwenden, wobei wir ziemlich sicher sein können, daß die Daten auch korrekt sind, wenn wir sie wieder zurücklesen. Auch wegen der Unzulänglichkeit des Bandtransports werden die Daten vom BIOS in Blöcken abgelegt, und zwar von fester Länge und nicht beliebiger. Die Routine muß nämlich immer eine bestimmte Zeit warten, um sicherzugehen, daß der Kassettenmotor eingeschaltet ist und die korrekte Geschwindigkeit einhält. Die Routine muß außerdem Synchronisationszeichen auf das Band schreiben, um sicherzustellen, daß der Prozessor die Signale auch wieder richtig synchronisiert, wenn die Daten vom Band gelesen werden sollen. Schließlich schreibt das BIOS noch ein CRC-Kontrollwort und einen Blocknachlauf im Anschluß an den Datenblock auf das Magnetband. All dies wird zusammen mit dem jeweiligen gewünschten Datenblock ausgegeben, ob er nun 1 Byte oder aber 10.000 lang ist. IBM wählte deshalb eine Blockgröße von 256, um damit einen Kompromiß zwischen diesem Blockoverhead und dem Magnetbandverbrauch zu finden.

Die beiden anderen Kassetten-BIOS-Funktionen sind einfache Steuerbits. Die Routinen dienen nämlich zum Ein- und Ausschalten des Kassettenmotors. Sollten Sie daran denken, ein eigenes, einfaches Gerät zum Anschluß für den IBM PC entwikkeln zu wollen, so bietet Ihnen der Kassettenport eine attraktive Methode zum Anschluß. Über den Kassettenanschluß an der Rückseite der Systemeinheit können wir eine serielle Eingangs- bzw. Ausgangsleitung bedienen. Es steht sogar eine Relaissteuerung zu unserer Verfügung, über die wir einen Schwachstrommotor mit geringer Stromaufnahme betreiben können. Auf eines müssen wir allerdings achten: Die Kassettenausgangsleitung ist direkt mit der Eingangsleitung verbunden, solange das Motorrelais nicht angezogen hat. Diese Verbindung gestattet es den IBM-Diagnoseroutinen, die Kassetteneingangs- und Ausgangsschaltkreise zu testen, ohne direkt Schreib- und Lesebefehle auf die Kassette ausführen zu müssen. Um jedoch den seriellen Eingang bzw. Ausgang getrennt zu benutzen, müssen wir den Motor einschalten — selbst wenn es keinen gibt.

Diskette

Die Disketten-BIOS-Routine ermöglicht Ein/Ausgabeoperationen auf Blockebene für den Diskettenadapter. Das BIOS-Interface gestattet es dem aufrufenden Programm dabei, für eine Disketten-Schreib/Leseoperation Spur- bzw. Sektoradressen anzugeben. Das BIOS steuert dann Adapter und Laufwerk für die gewünschte Operation und transportiert die Daten von oder nach dem vom Benutzer definierten Puffer. Nach Befehlsausführung faßt das BIOS die Ergebnisse der Operation zusammen und stellt sie dem Benutzer als Teil des Ausgabeparameters wieder zur Verfügung. Das aufrufende Programm muß sich also nicht mit den verschiedenen Funktionen der Diskettensteuerung und eventuellen zeitkritischen Einzelheiten befassen.

Disketten-Datenbereiche

Der Datenbereich des Disketten-BIOS beginnt bei Offset 3EH im BIOS DATA-Segment. Die ersten vier Bytes dieses Bereichs enthalten Statusinformationen über das Diskettenlaufwerk, die von Befehl zu Befehl weiter verwendet werden. Der 7-Byte Puffer mit Namen NEC_STATUS enthält Statusinformationen, die der NEC-Diskettenkontroller nach einer Schreib- oder Leseoperation zurückgibt. Wie wir aus dem Steuerprogramm entnehmen können, erlaubt es dieser Pufferbereich dem BIOS, auftretende Fehlerbedingungen mit einem einfachen Satz von Fehlercodes darzustellen. Das BIOS legt diese Fehlercodes im Byte DISKETTE_STATUS ab und gibt den Code außerdem im AH-Register an das aufrufende Programm zurück. Die Gleichsetzungen nach DISKETTE_STATUS fassen die Fehlercodes für das aufrufende Programm noch einmal zusammen.

Die NEC-Diskettensteuerung kennt zu allen Zeiten die Position des Schreib/Lesekopfes in jedem der vier Laufwerke, die sie bedienen kann. Allerdings muß die Steuerung mit dem jeweiligen Laufwerk synchronisiert sein, bevor sie die genaue Position des Kopfes ermitteln kann. So weiß die Steuerung z.B. nicht, wo sich die Köpfe gerade befinden, wenn der Strom eingeschaltet bzw. der Diskettenkontroller rückgesetzt wurde. Das Byte SEEK_STATUS verwendet die vier niedrigwertigen Bits, eines pro Laufwerk, um anzuzeigen, ob dem Kontroller die aktuelle Position des Kopfes für das jeweilige Laufwerk bekannt ist. Das Disketten-BIOS setzt dieses Byte auf 0, wenn es den Kontroller rücksetzt. Für jede weitere Diskettenoperation prüft das BIOS das Byte vor Ausführung eines Befehls. Ist das dem Laufwerk entsprechende Bit 0, so wird vor dem Suchbefehl ein Rekalibrierungsbefehl an das Laufwerk gesandt. Dieser Befehl positioniert den Schreib/Lesekopf auf Spur 0, womit bei Steuerung und Laufwerk wieder die gleiche Kopfposition entsteht. Für alle weiteren Suchbefehle kommt das Laufwerk dann ohne Rekalibrierung aus.

Im normalen Betrieb stellt die Rekalibrierung außerdem ein wertvolles Mittel zur Fehlerbehebung dar. Die empfohlene Aktion nach jeder Art von Diskettenfehler ist nämlich das Rücksetzen des Kontrollers. Dies stellt sicher, daß die Fehlerbedingung gelöscht ist, und erzwingt außerdem ein Rücksetzen von SEEK_STATUS. So wird das Laufwerk vom BIOS rekalibiert, bevor das Anwenderprogramm den Befehl erneut versucht (eine mißlungene Operation sollte dreimal wiederholt werden, da die meisten Diskettenfehler sporadisch sind — sie wiederholen sich nicht). So werden Diskettenfehler wie das Positionieren auf die falsche Spur automatisch behoben. Das Rekalibrieren und erneute Aufsuchen der gewünschten Spur löscht normalerweise den aufgetretenen Fehler.

Die Bytes MOTOR_STATUS und MOTOR_COUNT steuern den Lauf des Diskettenmotors. Der Diskettenadapter verfügt über ein Steuerregister, daß uns die Auswahl des gewünschten Motors erlaubt. In dieses Steuerregister kann nur geschrieben werden, weshalb das Byte MOTOR_STATUS als Speicherabbild des Registers dient. Vor einem Schreibbefehl müssen wir nämlich dafür sorgen, daß der Diskettenmotor die korrekte Umdrehungszahl erreicht hat, wozu wir einige Zeit, etwa eine halbe Sekunde, warten müssen. Läuft der Motor bereits, entfällt diese Wartezeit. Das Lesen des entsprechenden Bits im Statuswort erlaubt es dem BIOS, festzustellen, ob ein solcher Wartezyklus benötigt wird.

Ähnlich kann es von Vorteil sein, den Motor nach einer Diskettenoperation noch für einige Zeit laufen zu lassen. Dies ist eine Wette darauf, daß das Laufwerk in Kürze wieder benötigt wird. Wir wägen also die Abnutzung des Laufwerks gegen den Zeitgewinn beim nächsten Diskettenzugriff ab. Da das BIOS die Diskette allerdings nicht für ewige Zeiten laufen läßt, enthält die Variable MOTOR_COUNT einen Zählwert. Jeder Zeitgeber-Tick erniedrigt diesen Zählwert. Bei Erreichen von 0 wird dann der Diskettenmotor abgeschaltet. Die Verwendung des Standardparameters erlaubt es dem Diskettenmotor, nach jeder Operation noch etwa zwei Sekunden weiterzulaufen.

Wenn wir uns das Disketten-BIOS genau ansehen, werden wir feststellen, daß als einer der ersten Schritte MOTOR_COUNT auf 255 gesetzt wird. Dies stellt sicher, daß der Zeitgeber-Tick den Motor nicht mitten in einer Diskettenoperation abschaltet. Der endgültige Wert, der etwa zwei Sekunden entspricht, wird vom BIOS erst bei Rückkehr in das aufrufende Programm eingesetzt.

Abbildung 9.4 faßt die Befehle für das Disketten-BIOS zusammen. Der Resetbefehl initialisiert den Kontroller mit Laufwerksparametern wie Schrittrate und DMA-Modus. Der Befehl führt außerdem einen Hardwarereset auf den Kontroller aus. IBM empfiehlt diese Aktion auch als Antwort auf jede Art von Diskettenfehler. Sie ist nötig, da einige der Fehler (besonders der Time-Out Fehler, der auftritt, wenn sich keine Diskette im Laufwerk befindet) den Kontroller wirklich verwirren können. Nach solchen Fehlern ist die einzige Möglichkeit, den Kontroller wieder richtig zum Laufen zu bringen, ihn rückzusetzen.

AH	Funktion
0	Rücksetzen Diskettenadapter
1	Lesen Status der letzten Operation
2	Lesen von Disketten in Speicher
3	Schreiben von Speicher auf Diskette
4	Prüfen Diskette
5	Formatieren einer Diskettenspur

Abbildung 9.4 Funktionen des Disketten-BIOS

Schreib- und Lesebefehle

Die Schreib- und Lesebefehle benützen die Register des 8088 als Eingabeparameter für die Angabe von Spur, Sektor und Kopf, sowie zur Spezifikation des Laufwerks, für das der Befehl ausgeführt werden soll. Das aufrufende Programm bestimmt dabei über das ES:BX-Registerpaar einen Pufferbereich, und das Disketten-BIOS steuert die DMA so, daß für die weiteren Operationen dieser Puffer verwendet wird. Das Unterprogramm DMA_SETUP führt die dafür nötige Adressberechnung durch. Die Routine berechnet die Gesamtzahl der zu übermittelnden Bytes anhand der Anzahl der Sektoren, die als Eingabeparameter übergeben wurde, und der Sektorgröße, die als Tabellenparameter zur Verfügung steht. Der errechnete Wert und die bestimmte Adresse werden dann an den DMA-Kontroller weitergeleitet. Halten wir

dabei fest, daß die Routine erkennt, wenn der Puffer eine 64K-Adressgrenze überschreitet. Da das 4-Bit Seitenregister nicht erhöht wird, wenn das DMA-Adressregister über 0FFFFH umklappt, werden falsche Daten übertragen, sobald der Puffer eine solche Grenze überschreitet. Die Routine setzt nun eine Fehlerbedingung, die eine weitere Diskettenoperation, da sie nicht mehr korrekt wäre, verhindert.

Prüfbefehl

Der Prüfbefehl entspricht dem Lesebefehl, abgesehen davon, daß die Daten nicht im Speicher abgelegt, sonden einfach weggeworfen werden. Auch für die DMA gibt es einen speziellen Prüfbefehl, wobei der DMA-Kontroller auf die entsprechenden Anforderungen des Diskettenkontrollers antwortet und für die korrekten Buszyklen sorgt. Doch die jeweils zweite Hälfte eines DMA-Zyklus, die zum Ablegen der Daten im Speicher dient, wird nicht ausgeführt. Wir können diesen Prüfbefehl dazu verwenden, festzustellen, ob die Daten korrekt auf die Diskette geschrieben wurden. DOS verwendet diesen Befehl als Teil des FORMAT-Kommandos. Dabei wird die Diskette auf etwaige defekte Stellen untersucht. Werden sie durch den Prüfbefehl entdeckt, kann DOS sie über das Inhaltsverzeichnis von der Verwendung ausschließen. So können wir Disketten mit kleinen Fehlern weiterverwenden, ohne sie sofort wegwerfen zu müssen.

Obwohl wir mit dem Prüfbefehl Fehler auf der Diskette feststellen können, ist dies noch keine Garantie dafür, daß die Daten auch wirklich auf die Diskette geschrieben wurden. Nehmen wir dazu einmal an, daß sich Probleme mit den Schreibschaltkreisen ergeben haben. Der Fehler liegt dabei so, daß während einer Schreiboperation kein Fehler auftritt, aber keine Daten auf die Diskette geschrieben werden. Prüfen wir diesen Datenbereich, so liest der Prüfbefehl die ursprünglich dort vorhandenen Daten (die wir eigentlich ändern wollten) ohne Fehler, und wir könnten annehmen, daß alles korrekt verlaufen ist. Wollen wir absolut sichergehen, daß sich unsere Daten tatsächlich auf der Diskette befinden, so müssen wir sie nach dem Schreiben in einen anderen Bereich lesen und diese beiden Pufferbereiche vergleichen. Erst dann haben wir die Garantie, daß die Daten vollständig und korrekt geschrieben wurden.

Formatierungsbefehl

Der Formatierungsbefehl initialisiert eine fabrikneue Diskette. Beim Formatieren schreiben wir die Sektorkennungen auf die Diskette. Der Kontroller benutzt diese Kennfelder bei Schreib- und Lesebefehlen zur Bestimmung der gewünschten Sektoren. Bei einer Leseoperation sendet das BIOS z.B. vier Bytes Sektoridentifikation an den Diskettenkontroller. Diese vier Bytes entsprechen normalerweise Spurnummer, Kopfnummer, Sektornummer und Sektorlänge (Cylinder-Head-Record-Number: CHRN). Der Kontroller verwendet diesen CHRN-Wert zum Vergleich mit dem beim Formatieren geschriebenen Sektorkennfeld.

Dies bedeutet, daß der Kontroller sich nicht darum kümmert, welche Werte sich wirklich im CHRN-Feld auf der Diskette befinden. Auch müssen die Sektoren nicht in

jeder Spur von 1 bis 8 durchnumeriert sein. Sobald der Kontroller einen Sektor findet, dessen Kennfeld dem CHRN-Wert entspricht, liest er ihn. Während des Formatierens werden diese CHRN-Werte vom Kontroller auf die Diskette geschrieben. Wir haben also die Möglichkeit, beliebige CHRN-Werte zu wählen. Der Datenbereich für den Formatierungsbefehl enthält die CHRN-Werte für jeden Sektor, der geschrieben werden soll. Normalerweise enthält dieser Bereich Werte wie

```
DB      10,0,1,2,10,0,2,2
DB      10,0,3,2,10,0,4,2
...
```

für z.B. Spur 10 auf Seite 0 der Diskette. Das wäre das Datenfeld, wie es von DOS beim FORMAT-Kommando verwendet wird. In Abbildung 9.5. sehen wir ein Programm, das eine einseitige Diskette mit normalen Werten formatiert. Sie sollten allerdings das FORMAT-Programm des DOS nicht durch Ihr eigenes ersetzen, da DOS die Diskette auch prüft und das Inhaltsverzeichnis sowie die Dateizuordnungstabelle schreibt. Ein weiteres Problem könnte darin bestehen, daß das Programm sofort damit beginnt, die Diskette im Laufwerk A: zu formatieren. Wenn Sie das Programm laufen lassen wollen, sollten Sie sich also darauf einstellen. Zumindest zeigt unser Programm jedoch, wie man das Disketten-BIOS anwenden kann.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 9.5  Diskette Format


1                                               PAGE    ,132
2                                               TITLE   Figure 9.5  Diskette Format
3       0000                            STACK   SEGMENT STACK
4       0000      40 [                          DW      64 DUP(?)
5                         ????
6                                 ]
7
8       0080                            STACK   ENDS
9
10      0000                            CODE    SEGMENT
11                                              ASSUME  CS:CODE,ES:CODE
12      0000    00 00 01 02 00 00       ID_BUFFER       DB      0,0,1,2,0,0,2,2
13              02 02
14      0008    00 00 03 02 00 00                       DB      0,0,3,2,0,0,4,2
15              04 02
16      0010    00 00 05 02 00 00                       DB      0,0,5,2,0,0,6,2
17              06 02
18      0018    00 00 07 02 00 00                       DB      0,0,7,2,0,0,8,2
19              08 02
20
21      0020                            FORMAT  PROC    FAR
22      0020    1E                              PUSH    DS
23      0021    2B C0                           SUB     AX,AX
24      0023    50                              PUSH    AX              ; Set return address
25      0024    2E: 8D 1E 0000 R                LEA     BX,ID_BUFFER    ; Set buffer address
26      0029    0E                              PUSH    CS              ;  in ES:BX
27      002A    07                              POP     ES
28      002B    B9 0001                         MOV     CX,1            ; Track 0. sector 1
29      002E    BA 0000                         MOV     DX,0            ; Drive 0, head 0
30      0031                            TRACK_LOOP:
31      0031    2E: 8D 3E 0000 R                LEA     DI,ID_BUFFER    ; Need to reset C value in ID_BUFFER
32      0036    B0 08                           MOV     AL,8
33      0038                            ID_SETUP:
34      0038    26: 88 2D                       MOV     ES:[DI],CH      ; Set value for this cylinder
35      003B    83 C7 04                        ADD     DI,4            ; Go to next value in table
36      003E    FE C8                           DEC     AL
37      0040    75 F6                           JNZ     ID_SETUP        ; Loop through ID_BUFFER
38      0042    B8 0501                         MOV     AX,501H         ; Format command
39      0045    CD 13                           INT     13H
40      0047    FE C5                           INC     CH              ; Move to next cylinder
41      0049    80 FD 28                        CMP     CH,40           ; Are we done yet?
42      004C    75 E3                           JNE     TRACK_LOOP      ; Move on to next track
43      004E    CB                              RET                     ; Return to DOS
44      004F                            FORMAT  ENDP
45      004F                            CODE    ENDS
46                                              END     FORMAT
```

Abbildung 9.5 Diskettenformatierung

Sie könnten den Formatierungsbefehl beispielsweise sehr vorteilhaft dafür einsetzen, Ihre Diskette mit einem Kopierschutz zu versehen. Kopierschutz bedeutet, daß Ihre Diskette auf eine Weise beschrieben wurde, die ein Kopieren zumindest sehr erschwert. Da das DISKCOPY-Programm von DOS davon ausgeht, daß die Sektorkennfelder der Diskette in der normalen Art und Weise geschrieben wurden, kann es keine Disketten kopieren, deren Sektorkennfelder von der Norm abweichen. Durch das Schreiben von Sektorkennfeldern abweichend von der Norm können wir also einen gewissen Kopierschutz für eine Diskette herstellen.

Als Beispiel wollen wir eine Diskette dadurch mit Kopierschutz versehen, daß wir auf Spur 10 ungewöhnliche Sektorkennungen schreiben. Unser obiges Beispiel zeigte die normalen Werte; stattdessen enthält der Puffer nun:

```
DB    10,0,10,2,10,0,2,2
DB    10,0,3,2,10,0,4,2
...
```

Spur 10 enthält also keinen Sektor 1, dafür aber einen Sektor 10, den wir auf keiner normalen DOS-Diskette finden werden. DISKCOPY wird die Spur 10 also nicht korrekt kopieren können. Wenn wir in unserem Programm nun (mit dem Prüfbefehl) auf Spur 10, Sektor 10, prüfen, wird uns ein korrektes Ergebnis anzeigen, daß es sich um die Originaldiskette und nicht um eine versuchte Kopie handelt.

Dieser Kopierschutz ist natürlich nicht narrensicher. Ein einfallsreicher Benutzer (und sogar einige Kopierprogramme) kann ihn herausfinden und umgehen. Auch können wir die Sektorkennfelder nicht ganz nach Belieben verändern. Das BIOS verwendet die Spurnummer zum Festlegen der Suchadresse, weshalb sie immer zum jeweiligen Sektor passen muß. Außerdem bestimmt der Wert im Byte für die Kopfnummer hardwaremäßig den jeweiligen Schreib/Lesekopf, weshalb auch dieser Wert korrekt sein muß. Das Längenfeld kommt aus einer Parametertabelle, wird also nicht über ein Register eingegeben, und ist deshalb auch schwieriger zu ändern. Auch verwenden sowohl das BIOS wie der Diskettenkontroller diesen Wert zum Ermitteln der Sektorlänge, so daß wir diesen Wert nur nach reiflicher Überlegung verändern sollten. Es bleibt nur noch die Sektornummer. Bevor wir sie verändern, sollten wir eines bedenken: Soll die Diskette weiterhin unter DOS verwendet werden, so wird DOS immer versuchen, den von uns mit einer neuen Nummer versehenen Sektor zu verwenden, solange wir nicht das Disketteninhaltsverzeichnis so verändert haben, daß dieser Sektor reserviert bleibt. Wenn Sie Lesebefehle über mehrere Sektoren ausführen wollen (die das BIOS gestattet), so müssen die Sektoren aufsteigend numeriert sein, jedoch muß die Numerierung nicht unbedingt bei 1 beginnen.

Zusammenfassend können wir sagen, daß der Formatierungsbefehl Ihnen ein gewisses Maß von Kopierschutz verschafft. Doch gibt es noch immer keinen absoluten Schutz. Eine vernünftige Verschlüsselungstechnik kann jedoch helfen, daß ehrliche Leute nicht unbedingt in Versuchung geführt werden.

Video

Das Video-BIOS steuert die Funktion der beiden Bildschirmadapter, die für den IBM PC erhältlich sind. Wir haben uns diesen Bereich bis zum Schluß aufgehoben, da er die größte und auch komplizierteste BIOS-Routine bildet.

Video-Datenbereiche

Dieser Abschnitt des ROM BIOS-Datenbereichs ist mit VIDEO DISPLAY DATA AREA bezeichnet. Er beginnt bei Offset 49H und enthält die Variablen, die von den Routinen des Video-BIOS verwendet werden. Alle Datenbereiche enthalten Werte zur dauernden Benutzung durch die Bildschirmsteuerkarten. Viele dieser Werte stehen dabei in write-only Registern der Adapterkarten. BIOS muß die aktuellen Werte solcher Variablen, wie CRT_MODE_SET oder CRT_PALLETTE, kennen, wenn es diese Register modifizieren soll. Im Gegensatz zum Ausgabeport (Port 61H) auf der Systemplatine kann das BIOS diese Register vor dem Ändern oder Rückschreiben aber nicht lesen. Es muß deshalb ein Speicherabbild der Register mitführen.

Alle Variablen sind mit erklärenden Kommentaren versehen, die von Nutzen sind, wenn Sie sich das BIOS und diese Datenbereiche genauer erarbeiten wollen. Das Array CURSOR_POSN verdient eine besondere Erwähnung. Da die Farbkarte im Textmodus mehr als eine Bildschirmseite speichern kann, gibt es für jede dieser Seiten eine Speicherstelle für die Cursorposition. Der 6845 CRT-Kontroller kann nämlich nur die aktuelle Cursorposition bearbeiten. Wenn das BIOS nun von einer Bildschirmseite auf die nächste wechselt, muß es die Cursorposition für jede Seite speichern. Da die Farbkarte im Textmodus mit 40 Zeichen/Zeile maximal acht Bildschirmseiten speichern kann, sind auch acht Speicherstellen für die Cursorposition reserviert.

Funktionen des Video-BIOS

Die Routinen für das Bildschirm-BIOS umfassen eine große Zahl verschiedener Funktionen. In Abbildung 9.6 sehen wir die einzelnen Funktionen, die das BIOS hier ausführen kann.

Da das Video-BIOS über so viele Funktionen verfügt, wird eine Sprungtabelle verwendet, um die einzelnen Funktionen anzusprechen. Die Tabelle M1 enthält dabei die Offsetwerte für die einzelnen Einsprungpunkte im Video-BIOS. Im ersten Teil der Routine VIDEO_IO wird der Inhalt des AH-Registers in eine Sprungadresse in den BIOS ROM verwandelt. Zuerst werden jedoch noch einige andere Kleinigkeiten erledigt, einschließlich der Prüfung der Variablen EQUIP_FLAG.

Das Video-BIOS wurde von IBM so konzipiert, daß es mit beiden Adapterkarten, also mit der Farb/Graphik-Karte und auch mit der Schwarz/Weiß-Karte arbeiten kann. Doch geht das BIOS davon aus, daß immer nur eine der beiden Karten in Betrieb ist. Wir können also das BIOS nicht dazu verwenden, ein Zeichen auf den Farbbildschirm und direkt darauf ein Zeichen auf den Schwarz/Weiß-Bildschirm auszugeben. Wir können jeweils nur einen Adapter ansprechen.

AH	Funktion
0	Initialisieren Bildschirmadapter
1	Setzen Größe und Form des Cursors
2	Setzen Cursorposition
3	Lesen Cursorposition
4	Lesen Position des Lichtstifts
5	Auswählen aktive Seite
6	Bildschirm nach oben verschieben
7	Bildschirm nach unten verschieben
8	Lesen Zeichen
9	Schreiben Zeichen und Attribut
10	Schreiben Zeichen ohne Attribut
11	Auswählen Farbpalette
12	Schreiben Dot (Punkt)
13	Lesen Dot
14	Schreiben im Fernschreibmodus

Abbildung 9.6 Funktionen des Video-BIOS

Bei jedem Aufruf des Video-BIOS wird zuerst festgestellt, welcher Adapter sich im System befindet. Dies geschieht durch Prüfen der Bits von EQUIP_FLAG – speziell der beiden Bits für den angeschlossenen Bildschirm. Stehen die Bits 5 und 4 auf „1", so ist die Schwarz/Weiß-Karte angeschlossen. Jede andere Bitkombination zeigt an, daß sich eine Farbkarte im System befindet. Das Programm wurde von IBM also in der Annahme geschrieben, daß immer nur ein Bildschirmadapter an das System angeschlossen ist. Wir müssen deshalb die entsprechenden Schalter auf der Systemplatine so einstellen, daß beim Einschalten des Systems der angeschlossene Bildschirmadapter erkannt werden kann.

Die Information in EQUIP_FLAG wird vom BIOS dazu verwendet, um die Adresse des zu benutzenden Bildschirmpuffers zu bestimmen. BIOS setzt das ES-Register für die Schwarz/Weiß-Karte auf 0B000H, für die Farbkarte auf 0B800H. Dies erlaubt es den anderen BIOS-Routinen, den Bildschirm anzusprechen, ohne zu wissen, welcher Adapter nun wirklich an das System angeschlossen ist. Alle Bezüge auf den Bildschirmpuffer erfolgen also relativ zum ES-Register.

Da EQUIP_FLAG den aktuellen Bildschirmadapter angibt, könnten wir nun annehmen, es würde genügen, einfach die Bits in diesem Flagwort zu ändern, um von einer Adapterkarte auf die andere umzuschalten. Dummerweise ist das aber nicht so. Die Portadresse des 6845 ist für die beiden Adapter unterschiedlich, und BIOS speichert die Basisadresse des Bildschirmadapters im BIOS-Datenbereich. Die Variable ADDR_6845 wird vom BIOS dabei nur bei der Initialisierung des Adapters (AH = 0) gesetzt. Ein Umschalten von einer Karte auf die andere erfordert also auch ein Setzen dieser Variablen.

Obwohl die Variable CURSOR_POSN über acht Speicherplätze verfügt, können wir damit nicht einfach den Wechsel von einem Adapter auf den anderen durchführen. Wir müssen die Cursorpositionen im BIOS-Datenbereich bei jedem Adapterwechsel erneut besetzen. Tun wir dies nicht, so wird die Cursorposition auf dem Bildschirm nicht mehr mit der im Datenbereich übereinstimmen. Wird dann ein Zeichen auf den Bildschirm ausgegeben, so erscheint es wahrscheinlich an der falschen Stelle.

IBM hat aus diesem Grund sogar selbst einige Methoden veröffentlicht, mit denen ein Wechsel von einer auf die andere Adapterkarte möglich wird. Die Möglichkeit besteht dabei für Assembler- und BASIC-Programme. Dazu ist es erforderlich, EQUIP_FLAG zu modifizieren, um den gewünschten Adapter anzuzeigen. Danach wird ein Video-Interrupt (INT 10H) mit AH=0 erzeugt. Diese Funktion initialisiert erneut den Bildschirmadapter und stellt sicher, daß alle BIOS-Variablen korrekt gesetzt sind. BIOS arbeitet von diesem Zeitpunkt an nur noch mit dem neu eingestellten Adapter. Alle bis dahin auf den vorherigen Adapter ausgegebenen Zeichen bleiben erhalten. Auch werden alle weiteren Veränderungen im Bildschirmpuffer dieses Adapters weiterhin ausgegeben. Wir können also durch direktes Ändern der Pufferinhalte ohne den Umweg über das BIOS weiterhin Informationen auf den Bildschirm ausgeben, von dem wir uns gerade weggeschaltet haben.

Verwenden wir dazu ein einfaches Beispiel. Wir haben einen IBM PC mit Schwarz/Weiß- und Farbkarte, und beide Karten sind angeschlossen. Schalten wir den Rechner ein, so verwendet das System automatisch den Schwarz/Weiß-Bildschirm. Wir müssen die Schalter auf der Systemplatine so einstellen, da der Schwarz/Weiß-Bildschirm beschädigt werden kann, wenn er nicht kurz nach dem Einschalten initialisiert wird. Im Bedienerhandbuch wird deshalb empfohlen, diese Schalter auf „Schwarz/Weiß" zu stellen.

Wir können nun das Video-BIOS mit der Schwarz/Weiß-Karte verwenden. Um auf die Farbkarte umzuschalten, benutzen wir das Programm aus Abbildung 9.7. Diese Routine schaltet auf den Farbbildschirm im 80-Zeichen Textmodus. Die bisher auf den Schwarz/Weiß-Bildschirm ausgegebenen Zeichen bleiben erhalten. Wir können nun das Video-BIOS mit der Farbkarte verwenden. Wollen wir dagegen Zeichen auf dem Scharz/Weiß-Bildschirm verändern, so können wir dies durch einfaches Schreiben der neuen Zeichen oder Attributwerte in den Bildschirmpuffer bei Segment 0B000H erreichen. Dadurch verändern wir zwar nicht die Position des

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 9.7  Switch to Color Card

1                                          PAGE    ,132
2                                          TITLE   Figure 9.7  Switch to Color Card
3       0000                       STACK   SEGMENT STACK
4       0000      40 [                     DW      64 DUP(?)
5                         ????
6                                ]
7
8       0080                       STACK   ENDS
9       0000                       ABS0    SEGMENT AT 0
10      0410                               ORG     410H
11      0410                       EQUIP_FLAG      LABEL   BYTE     ; Only will modify low byte of flag
12      0410                       ABS0    ENDS
13      0000                       CODE    SEGMENT
14                                         ASSUME  CS:CODE,DS:ABS0
15      0000                       COLOR   PROC    FAR
16      0000  1E                           PUSH    DS
17      0001  2B C0                        SUB     AX,AX            ; Return address
18      0003  50                           PUSH    AX
19      0004  8E D8                        MOV     DS,AX            ; Set up ABS0 segment
20      0006  80 26 0410 R CF              AND     EQUIP_FLAG,11001111B    ; Zero out display field
21      000B  80 0E 0410 R 20              OR      EQUIP_FLAG,00100000B    ; Indicate 80 column color card
22      0010  B8 0003                      MOV     AX,3             ; Mode set for 80 column color
23      0013  CD 10                        INT     10H              ; Reset the card
24      0015  CB                           RET                      ; All done
25      0016                       COLOR   ENDP
26      0016                       CODE    ENDS
27                                         END
```

Abbildung 9.7 Umschalten auf die Farbkarte

Cursors, aber den Inhalt des Bildschirms. Wollen wir bei diesem Beispiel gleichzeitig auf beide Bildschirme schreiben, so müssen wir unsere eigene Routine zur Versorgung der Schwarz/Weiß-Karte schreiben. Oder Sie versuchen herauszufinden, welche Werte im Datenbereich des Video-BIOS verändert werden müssen, damit das BIOS zwischen beiden Karten wechseln kann, ohne diese jedes Mal neu zu initialisieren.

Setzen des Bildschirmmodus

Gibt ein Programm den Interrupt INT 10H mit AH=0 aus, so wird damit die Modusfunktion des Video-BIOS aufgerufen. Zeigt EQUIP_FLAG dabei an, daß eine Schwarz/Weiß-Karte angeschlossen ist, so ist der Wert im AH-Register ohne Bedeutung. In diesem Fall setzt BIOS den Schwarz/Weiß-Adapter auf Modus 7, was dem Textmodus mit 80x25 Zeichen entspricht.

Bei der Farbkarte bestimmt der Wert in AL jedoch, auf welchen der beiden Text- bzw. Graphikmodi die Karte gesetzt werden soll. Wir sehen dabei, daß es die verschiedenen Text- bzw. den 320x200 Graphikmodus sowohl in Farbe als auch in Schwarz/Weiß gibt. Die Schwarz/Weiß-Versionen schalten im Grunde die Farbe nicht aus. Sie unterdrücken nur das Burst-Signal, das ein Farbfernsehgerät benötigt, um die Farbe der einzelnen Bildpunkte zu bestimmen. Benützen Sie also einen Farbmonitor, so werden die Farben immer noch vorhanden sein. Schalten wir bei einem Farb- bzw. Schwarz/Weiß-Monitor in Verbindung mit der Farbkarte die Farben aus, so erhalten wir ein geringfügig schärferes Bild. Benötigen wir in einem Programm keine Farben, so erzeugen wir ein besseres Bild, wenn wir anstelle der Farbmodi die Schwarz/Weiß-Modi einstellen.

Läuft die Routine zum Setzen des Bildschirmmodus ab, so werden der Adapter und die BIOS-Datenbereiche für die gewünschte Verarbeitung vorbereitet. Der Bildschirmpuffer wird dazu mit Blanks gefüllt und der Cursor auf die linke obere Ecke des Bildschirms positioniert. Solange Sie sich mit der Hardware nicht sehr genau auskennen, sollten Sie zum Setzen des Modus immer das BIOS verwenden. Obwohl Sie nämlich mit den einzusetzenden Werten eigentlich nichts falsch machen können, ist es doch sehr schwer, in einem Programm Fehler zu suchen, das den Bildschirm bedient. Im Falle eines Fehlers kann es vorkommen, daß der Bildschirm nicht mehr reagiert, und dann gibt es keine Möglichkeit mehr, herauszufinden, was wirklich schief ging.

Die BIOS-Routinen für die AH-Werte von 1 bis 5 dienen zum Verändern der Register des 6845. Wie wir aus dem Abschnitt über die Systemhardware wissen, verfügt der 6845 über eine Anzahl von Registern, die Größe und Aussehen des Cursors sowie einige Zeitsignale für den Bildschirm bestimmen. Über die Routinen des Video-BIOS können wir die Bildschirmausgaben verändern, ohne die Basisadresse des 6845 zu verwenden. Sie sind also vorwiegend als Serviceroutinen für unsere Zwecke gedacht.

Scrolling

Die Scrollroutinen bewegen die Textinformation auf dem Bildschirm je nach Routine auf- bzw. abwärts. Sie bieten uns damit eine Art Fenster — d.h. BIOS verschiebt nicht den ganzen Text. Der Parameter für die Scrollroutine bestimmt nur einen rechteckigen Bildausschnitt. Dazu geben wir die obere linke und untere rechte Ecke des gewünschten Ausschnitts an. Innerhalb dieses Bereichs wird der Text dann wirklich bewegt. Der restliche Bildschirminhalt bleibt unverändert.

Wir können die Verwendung solcher Fenster sehen, wenn wir betrachten, wie DOS und BASIC über das BIOS den Bildschirm auf- und abbewegen. Gehen wir dabei von einem 80-Zeichen Bildschirm aus, so setzt DOS die linke obere Ecke als (0,0) und die rechte untere Ecke als (24,79). Dadurch wird immer der gesamte Bildschirminhalt verschoben. BASIC dagegen verwendet die unterste Zeile für Statusinformationen und benützt sonst nur die oberen 24 Zeilen des Bildes. Ruft BASIC nun zum Verschieben des Bildschirminhalts das BIOS auf, so wird die linke obere Ecke wie vorher mit (0,0) angegeben, die rechte untere Ecke jedoch mit (23,79). Da die letzte Zeile nun außerhalb des Fensters liegt, wird sie auch nicht verschoben. Das nächste Kapitel zeigt ein Beispiel für ein solches Fenster, das von einem BASIC-Programm verwendet wird.

Das Video-BIOS führt die Verschiebeoperation durch Bewegen der Zeichen und Attribute im Bildschirmpuffer durch. Die Startadresse des Puffers wird also nicht verändert. Diese Methode wäre zwar erheblich schneller, sie würde es aber dem Anwenderprogramm nicht mehr ermöglichen, die Position der Zeichen auf dem Bildschirm festzulegen. Vom Zeitverhalten her entspricht die Verschiebeoperation des BIOS den Bedürfnissen einer normalen Bildschirmverarbeitung. Halten wir fest, daß die Routine auf Wunsch auch über mehr als eine Zeile verschieben kann. Normalerweise wird der Bildschirminhalt von einem Programm fast immer nur um eine Zeile weiterbewegt. BIOS gestattet uns aber auch ein Bewegen der Bildinhalte über mehrere Zeilen. Wird der Parameter auf 0 gesetzt, so wird der Bildschirm gelöscht. Wir haben damit einen einfachen Weg, den ganzen oder einen Teil des Bildschirms zu löschen.

Arbeitet ein Programm mit der Farbkarte im 80-Zeichen-Modus, so können wir nicht zu jeder beliebigen Zeit Daten in den Bildpuffer schreiben bzw. daraus lesen. Warten wir dazu nicht einen bestimmten Zeitpunkt ab, so können wir auf dem Bildschirm „Schnee" erzeugen. Da die Scrollroutine große Datenmengen schreibt bzw. liest, muß eine Vorkehrung für solche mögliche Interferenzen getroffen werden. Sehen wir uns den entsprechenden Abschnitt im BIOS an, so können wir erkennen, daß die Farbkarte im 80x25 Modus (CRT_MODE = 2 oder 3) als Sonderfall behandelt wird. Für Verschiebeoperationen wartet das BIOS nämlich hier bis zum Bildrücklauf. Zu diesem Zeitpunkt befindet sich der aktuelle Pufferinhalt vollständig auf dem Bildschirm, und die Elektronik kehrt zurück zum Bildanfang (Dieser Vorgang wiederholt sich 60 Mal pro Sekunde). Während des Bildrücklaufs schaltet das BIOS den Bildschirm völlig ab und führt die Verschiebeoperation durch. Sind alle gewünschten Zeichen verschoben, wird der Bildschirm wieder eingeschaltet. Dies verursacht ein kurzes Blinken des Bildes. Sehen Sie sich den Bildschirm während des Verschiebens aus der Nähe an, so werden Sie entdecken, daß die obersten

sechs Zeilen etwas schwächer erscheinen. Dies rührt aus der Tatsache, daß die Verschiebeoperation geringfügig länger dauert als ein Bilddurchlauf. So werden also die obersten sechs Zeilen für zwei Bilddurchläufe abgeschaltet, der Rest nur für einen. Diese Methode ist in jedem Fall dem Zulassen von Interferenzen auf dem Bildschirm vorzuziehen. Um den Unterschied herauszufinden, können Sie ja einmal ein eigenes Programm dafür schreiben.

Ein weiterer Teil der Scrollroutine kommt zum Zuge, wenn wir den Bildschirm im Graphikmodus betreiben. Obwohl dies ein wertvoller Teil des ROM BIOS ist, soll er hier nicht näher beschrieben werden. Wir werden später darauf eingehen, wenn wir besprochen haben, wie Zeichen auf den Bildschirm geschrieben bzw. von dort gelesen werden.

Schreiben und Lesen von Zeichen

Die Routinen des Video-BIOS mit Werten zwischen 8 und 10 im AH-Register bearbeiten Zeichen auf dem Bildschirm. Die drei Routinen verwenden dabei immer die aktuelle Cursorposition. Um ein Zeichen also an eine bestimmte Position auf dem Bildschirm zu schreiben, müssen wir zuerst mit dem Video-BIOS den Cursor an die gewünschte Stelle positionieren (AH=2). BIOS läßt den Cursor nach dem Schreiben eines Zeichens auch nicht automatisch weiterrücken. Wollen wir also mehr als ein Zeichen ausgeben, so müssen wir jeweils selbst den Cursor auf die neue Position bringen. Das gleiche gilt für das Lesen von Zeichen vom Bildschirm. Da alle Zeichenoperationen sich auf die aktuelle Cursorposition beziehen, brauchen wir bei Verwendung des Video-BIOS die Adresse des Zeichens im Bildpuffer nicht zu kennen. Die Schreib- und Leseroutinen finden für uns die jeweilige Adresse im Bildpuffer heraus. Wir müssen dazu nur Spalte und Zeile des gewünschten Zeichens auf dem Bildschirm angeben.

Es gibt zwei Arten von Schreibroutinen. Die eine benötigt Zeichen und Attribut (also Blinken, hohe Intensität, Farbe etc.) für die aktuelle Cursorposition. Die andere schreibt nur das Zeichen und verändert das Attribut nicht. Die beiden Möglichkeiten bestehen, um es einem Programm zu gestatten, mit vorhandenen Attributen weiterzuarbeiten, ohne diese zu kennen. Die Lesefunktion gibt dagegen Zeichen und Attribut der aktuellen Cursorposition zurück. Hier benötigen wir keine zwei getrennten Funktionen, es steht ja in unserem Belieben, die gelieferte Information auszuwerten.

Wegen der Möglichkeit von Interferenzen bei der Farbkarte verfügen die Schreib/Leseroutinen über einen Test auf Zeilenrücklauf. Dieser Test ist nötig, um sicherzustellen, daß während einer Operation kein „Schnee" auf dem Bildschirm auftritt. BIOS führt diesen Test immer aus, wenn ein Zeichen geschrieben werden soll, ohne Rücksicht auf den eingestellten Modus des Bildschirms. Der Test wird sogar bei der Schwarz/Weiß-Karte durchgeführt. Die maximale Wartezeit auf den Zeilenrücklauf beträgt dabei etwa 63 Mikrosekunden. Diese Wartezeit fällt bei der Ausgabe eines Zeichens auf den Bildschirm praktisch nicht ins Gewicht. Da im ROM nicht mehr genügend Platz war, ließ IBM den Test auf Sonderfälle hier einfach weg. So prüft das BIOS immer beim Schreiben und Lesen auf den Zeilenrücklauf.

Text im Graphikmodus

Eine der entscheidenden Eigenschaften des IBM ROM BIOS ist die Fähigkeit, Text auf den Bildschirm auszugeben, auch wenn dieser sich im Graphikmodus befindet. Das BIOS führt dies mit Hilfe der Zeichentabelle bei Offset 0FA6EH durch. Diese Tabelle enthält das Bitmuster für die ersten 128 Zeichen. Ein Benutzer kann den Interruptvektor 01FH so setzen, daß er auf eine Tabelle für die zweiten 128 Zeichen zeigt.

Wie wir aus der Liste des BIOS sehen können, verzweigt die Schreibroutine in einen besonderen Bereich GRAPHICS_WRITE, wenn die Farbkarte im Graphikmodus betrieben wird. Hier wird das Bitmuster aus der vom Benutzer gelieferten bzw. aus der Tabelle im ROM entnommen und punkteweise in den Bildschirmpuffer eingesetzt. Dabei geschehen einige interessante Dinge. Im mittelauflösenden Graphikmodus erweitert BIOS das 8-Bit Zeichenmuster auf 16 Bit. Dies erfolgt, da im 320x200 Graphikmodus pro Pixel zwei Bits zur Verfügung stehen. Das Unterprogramm S21 (EXPAND_BYTE) nimmt die Punktwerte in AL und erweitert sie zu einem ganzen Wort in AX.

Die Schreibroutine muß außerdem in der Lage sein, das Ansprechen der geradzahligen bzw. ungeradzahligen Adressen der Farbkarte durchzuführen. Bei GRAPHICS_ WRITE legt BIOS die einzelnen Punktreihen eines Zeichens von 2000H an ab. Dies ist besonders aus der Schreibroutine für hochauflösende Graphik zu erkennen. Hier kann BIOS das Zeichen direkt in den Bildschirmpuffer speichern. Doch anstelle eines Befehls REP MOVSB über die acht Bytes des Zeichenmusters finden wir bei S4 eine Schleife, die die einzelnen Adressen anspricht. Zuerst wird dabei mit STOSB an eine geradzahlige Adresse geschrieben. Dann wird das erste Feld mit einer ungeradzahligen Adresse mit MOV nach [DI+2000-1] belegt.

Eine weiterer Vorzug der Schreibroutine ist die Fähigkeit, Zeichen mit einer XOR-Funktion auf den Bildschirm auszugeben. Diese Methode ist von fast unschätzbarem Wert, wenn wir Zeichen auf den Graphikbildschirm ausgeben wollen, die später wieder verschwinden sollen. Schreibt BIOS ein Zeichen auf den Bildschirm, und das XOR-Bit ist gesetzt, so werden die aktuellen Inhalte des Bildpuffers mit dem zu schreibenden Bildmuster logisch mit XOR verknüpft. Normalerweise wird dadurch ein Zeichen lesbar ausgegeben, doch hängt das Aussehen des Zeichens in der Tat vom Hintergrund ab, auf dem es geschrieben wird. Schreibt BIOS nun dasselbe Zeichen mit der XOR-Funktion ein zweites Mal an die gleiche Stelle, so verschwindet dieses Zeichen und der Bildschirm zeigt wieder den ursprünglichen Inhalt. Diese Methode ist in jedem Fall dem einfachen Schreiben von Zeichen mit einem Löschen durch Überschreiben mit Blanks vorzuziehen. Das Überschreiben mit Blanks kann nämlich den ursprünglichen Bildschirminhalt nicht wiederherstellen. Wir können diese Fähigkeit sehr effektiv einsetzen, wenn wir kurzzeitig Texte in ein Bild einblenden wollen.

Auch die Leseroutine arbeitet korrekt, wenn sich die Farbkarte im APA-Modus befindet. BIOS entnimmt dabei das Bild des gewünschten Zeichens aus dem Bildpuffer und vergleicht es mit den Bildern in der Zeichentabelle. Passen die Bilder aufeinander, so gilt das Zeichen als erkannt. Allerdings arbeitet diese Routine nur mit einem vollständigen Bild des gewünschten Zeichens. Wird also das gewünschte Zeichen

beispielsweise von einer graphischen Darstellung berührt, so kann das Zeichen nicht mehr korrekt erkannt werden. Doch selbst mit dieser Einschränkung haben wir hier immer noch die Möglichkeit, sogar im APA-Modus Text zu verarbeiten. So lange unser Programm auf BIOS-Ebene arbeitet, können wir Text also unabhängig vom eingestellten Bildschirmmodus verarbeiten.

Wir erinnern uns daran, daß auch die Scrollroutinen über spezielle Abschnitte zur Behandlung von Graphik verfügten. Wenn wir uns diesen Abschnitt des BIOS noch einmal ansehen, können wir feststellen, daß BIOS auch auf dem Graphikbildschirm Fenster erkennen und dabei Zeichen wie im Textmodus verschieben kann. Allerdings geschieht das Verschieben im Graphikmodus etwas langsamer als im Textmodus, hauptsächlich deshalb, da die Routine statt der 2000 oder 4000 Bytes im Textmodus nun den ganzen, 16000 Byte langen Bildschirmpuffer bewegen muß. Dies erhöht die Verschiebezeit logischerweise um das Vier- bis Achtfache.

Die Fähigkeit des BIOS, auch im Graphikmodus Zeichen zu verarbeiten, verschafft uns ein großes Maß an Flexibilität. Es ist also kein Problem, Graphik oder Bilder zu erstellen, und diese dann im Textmodus zu beschriften. In ähnlicher Weise können wir auf einem Teil des Bildschirms Graphik darstellen und einen anderen Teil zur Ausgabe von Texten verwenden. Wir können in diesem Bereich mit den ganz gewöhnlichen Schreib- und Leseroutinen arbeiten, und wir können den Text nach Belieben verschieben. Wir können sogar Text auf den Bildschirm ausgeben, ohne uns um den eingestellten Bildschirmmodus zu kümmern. Das BIOS stellt den Bildschirmmodus für uns fest und schreibt korrekt die gewünschten Zeichen.

Graphik

Das Video-BIOS verfügt noch über einige zusätzliche Funktionen, die Ihnen bei der Darstellung von Graphik auf dem IBM PC helfen sollen. Durch Setzen des AH-Registers auf 11 kann ein Programm Farben für die Graphikdarstellung auswählen. Die Routine ist so aufgebaut, daß sie eine echte Farbauswahl anstelle der vorgegebenen Farbpalette der Farbkarte bietet. Könnten wir im 320x200 APA-Modus eine echte Farbauswahl treffen, so müßten wir aus den möglichen Farben vier für die vom Programm verwendete Palette wählen. Mit der Hintergrundfarbe im mittelauflösenden Graphikmodus ist dies möglich. Wir können jede beliebige Farbe als Farbe 0, die Hintergrundfarbe, auswählen. Die verwendete Routine gestattet also eine echte Farbauswahl, falls IBM irgendwann die Hardware dafür entsprechend modifiziert.

Wir legen nun den passenden Wert für das gewünschte Pixel im BH-Register ab. BL enthält dann die Farbe, die der Adapter für das Pixel erzeugen soll. Ist z.B. BH=0, dann enthält BL den Farbwert für den Hintergrund. Die BIOS-Routine behandelt dabei nur die Werte 0 und 1 im BH-Register, da wir nur die Hintergrundfarbe bzw. eine der beiden vordefinierten Paletten auswählen können. Der Prolog im BIOS enthält die Farbwerte für jede Palette. Wir können übrigens auch die Rahmenfarbe für den Textmodus über eine BIOS-Funktion bestimmen.

Die beiden anderen Graphikroutinen gestatten es uns, ein einzelnes Pixel auf dem Graphikbildschirm zu lesen bzw. zu schreiben. Für einfache Operationen genügt dabei dem BIOS die Angabe von Zeile und Spalte. Für große Figuren und alle anderen Arten von graphischen Darstellungen sind diese Operationen übrigens sehr zeitaufwendig. Wir müssen die BIOS-Routine nämlich für jeden Punkt auf dem Bildschirm aufrufen. Im hochauflösenden Graphikmodus mit 640x200 Punkten müßte unser Programm das BIOS 128000 Mal aufrufen, um den Bildschirm völlig zu beschreiben. Obwohl die BIOS-Routine sehr schnell abläuft, muß doch jedesmal die Speicheradresse über die Zeilen- und Spalteninformation errechnet werden. Dies erfordert eine Multiplikation und mehrere Additionen, und benötigt doch einige Zeit. Im allgemeinen erzeugen wir graphische Figuren deshalb über einen Startpunkt, und dann nur noch mit Offsetwerten zu diesem Punkt. Beim ersten Punkt wird also die Speicheradresse tatsächlich berechnet, während für alle weiteren Operationen nur noch entsprechende Offsetwerte addiert werden müssen.

Fernschreibmodus

Die Fernschreibfunktion des BIOS ist für Programme gedacht, die auf einfachste Art Daten auf den Bildschirm ausgeben wollen. Dabei wird der Bildschirm wie ein Fernschreiber behandelt. Die Positionierung des Cursors wird ebenso wie die Zeichenausgabe direkt vom BIOS vorgenommen. Nachdem BIOS ein Zeichen an die aktuelle Position geschrieben hat, bewegt es den Cursor auf die nächste Stelle. Erreicht der Cursor dabei das Ende einer Zeile, so wird das Bild um eine Zeile nach oben geschoben und die Ausgabe beginnt erneut am Zeilenanfang.

Neben der Tatsache, daß wir damit über eine einfache Methode zur Zeichenausgabe auf dem Bildschirm verfügen, kann die Fernschreibroutine auch als gutes Beispiel für die Zeichenbearbeitung durch das Video-BIOS dienen. Die Routine schreibt Zeichen, positioniert den Cursor, und führt, falls nötig, auch das Verschieben des Bildschirminhalts durch. Auch reagiert die Routine auf einige Steuerzeichen. Backspace bewegt den Cursor um eine Position zurück, Wagenrücklauf setzt den Cursor auf den Anfang der nächsten Zeile, und Zeilenvorschub führt nötigenfalls auch ein Verschieben des Bildschirms durch. Schließlich erzeugt der Steuercode BELL (ASCII-Zeichen 7) auch noch einen Ton auf dem Lautsprecher. DOS verwendet diese BIOS-Funktion übrigens für fast alle Ausgaben.

10 Erweiterungsroutinen und Unterprogramme in Assembler

Dieses Kapitel erläutert die Anwendung von Assemblerprogrammen im Rahmen größerer Programme. Alle bisherigen Beispiele waren nämlich selbständige Programme. Mit keiner anderen Sprache können wir so in die Hardware eingreifen wie mit dem Assembler. Doch ist die Assemblersprache in vielen Fällen sicherlich nicht die beste Wahl. Hier ist dann ein Programm in einer höheren Programmiersprache mit Assemblerunterprogrammen am Platz.

Im folgenden erläutern wir nun zwei Bereiche, in denen Assemblerprogramme benutzt werden. Der erste Fall sind Assemblerroutinen zur Erweiterung des ROM BIOS. Diese Routinen ermöglichen neue Funktionen für die vorhandene Hardware. Vor ihrer Benutzung müssen wir diese Routinen im Speicher resident machen. Dann können wir sie als Erweiterungen der normalen BIOS-Funktionen ansprechen. Wir werden dafür zwei Beispiele zeigen, die außerdem zwei verschiedene Methoden zum Laden der Routinen in den Speicher verwenden.

Der zweite Fall sind Assemblerunterprogramme für höhere Programmiersprachen. In diesem Fall führen wir mit dem Programm in Maschinensprache Dinge aus, die mit einer höheren Programmiersprache nur schwierig oder überhaupt nicht zu bewältigen sind. In mehreren Beispielen werden wir die verschiedenen Techniken dafür zeigen. Dabei geht es hauptsächlich um zusätzliche Funktionen der Hardware, doch ist das Konzept auf jede Problemlösung übertragbar.

Erweiterungen des BIOS

Das Programm für einige neue Gerätetreiberfunktionen wollen wir in den Speicher laden, damit es zu einer dauernden Erweiterung des Systems wird. Das ROM BIOS ist ein gutes Beispiel für einen solchen Typ von Programm. IBM machte die Gerätetreiber zu einem festen Bestandteil des Systems, indem sie im ROM abgelegt wurden. Da das BIOS immer vorhanden ist, sind diese Routinen für jedes Anwenderprogramm, das auf dem IBM PC abläuft, verfügbar.

Für die meisten von Ihnen wird es allerdings keine Möglichkeit geben, ein Programm in einem eigenen ROM abzulegen. Es hat auch wenig Sinn, eine große Summe Geld für die Herstellung eines einzigen ROM-Bausteins auszugeben, wenn wir unser Programm nicht sehr weit verbreiten oder vermarkten wollen. Allerdings gibt es eine wesentlich billigere Alternative. Es gibt einige ROM-Typen, die Sie selbst programmieren können, soweit Sie über das nötige Werkzeug verfügen. Etliche Firmen bieten sogenannte PROM-Programmer (Geräte zum Beschreiben von „Programmable ROMs") an, mit denen wir unsere eigenen ROM-Module herstellen können. Die nötige Hardware zum Erstellen von PROMs kostet nur noch wenige Hundert Mark, und einzelne PROMs sind bereits für 30-50 Mark erhältlich.

Für einige Programme könnte eine derartige Speichertechnik nötig sein, und in der Tat verfügt der IBM PC zu diesem Zeck über einen leeren ROM-Sockel. Wir können hier also einen Standard 8K-ROM oder PROM einsetzen. Dadurch wird unser Pro-

gramm zu einem festen Bestandteil des Rechners. Wir wollen die Herstellung von ROMs und PROMs aber nicht näher besprechen. Dafür ist nämlich besondere Hardware und meist für jeden Benutzer spezielles Vorgehen erforderlich.

Stattdessen wollen wir uns eine Möglichkeit ansehen, Programme so in den Schreib/Lesespeicher zu laden, daß sie zu einem festen Bestandteil des Systems werden. Ihr Programm bleibt dann bis zum Abschalten des Rechners speicherresident. Der Vorteil dieses Verfahrens liegt darin, daß das Programm nicht fest in den Rechner eingebaut werden muß. Wir können es also modifizieren, ohne extra den Rechner zu zerlegen. Finden wir eine Schwachstelle im Programm, so können wir dieses verändern, ohne all die Schritte zur Herstellung eines ROMs zu durchlaufen.

DOS-Programmende ohne Speicherlöschen

Der erste Weg, ein Programm zu schreiben und im Speicher resident zu halten, ist das DOS-Programmende ohne Speicherlöschen. Diese Funktion wird über INT 27H angesprochen.

Das normale Programmende in DOS ist INT 20H. Eine andere Möglichkeit wäre noch ein Sprung auf die Stelle 0 im Programmsegment-Präfix, wie wir es mit unseren .EXE-Programmen bereits getan haben. Dadurch wird die Steuerung zurück an DOS übergeben. DOS gibt dann den Speicherbereich, den es dem Programm zugewiesen hatte, wieder frei. Das nächste Programm, das wir nach INT 20H laden, wird also wieder denselben Platz wie das vorausgehende belegen.

Das Programmende über INT 27H läuft dagegen anders ab. Die Steuerung geht zwar wie bei INT 20H wieder an DOS zurück, doch ein Teil des vom Programm belegten Speichers wird nicht zur Wiederverwendung freigegeben. Das DX-Register muß dabei die Endadresse des gewünschten zu reservierenden Speicherbereichs enthalten. DOS kennzeichnet diesen Bereich dann als zum System gehörend. Das bedeutet, daß unser Programm nun zu einem Teil von DOS wird. Es gibt keinen Weg mehr, diesen Speicher freizugeben, als DOS erneut von Anfang an zu starten.

Wenn wir den Programmausgang INT 27H benutzen, muß das CX-Register die Adresse des Programmsegment-Präfixes enthalten. Der beste Weg hierzu ist ganz einfach alle Programme, die INT 27H verwenden, als .COM-Programme zu schreiben. Es ist nämlich ziemlich schwierig, ein .EXE-Programm zu schreiben, das bei Programmende das CS- und DX-Register korrekt gesetzt läßt. Da wir das Erstellen eines .COM-Programms bereits in Kapitel 5 besprochen haben, gehen wir einfach davon aus, daß alle unsere residenten Programme als .COM-Programme ablaufen.

Das Beispiel für INT 27H ist einigermaßen kompliziert. Wir sehen darin nämlich nicht nur die Anwendung von INT 27H, sondern auch eine Methode zum Ersetzen des aktuellen ROM BIOS durch eine neue Version. Wir werden sogar einige Tricks mit dem Zeitgeber anwenden, um die Ausführungsgeschwindigkeit zu erhöhen.

In Abbildung 10.1 sehen wir dieses Beispiel. Es verfügt über einen Druckpuffer. Normalerweise verwendet ein Programm zu jeder Druckausgabe eines Zeichens INT 17H, den ROM BIOS-Druckertreiber. Diese Funktion übergibt jeweils ein Zeichen an den Drucker, nachdem sie auf Fehler und die Bereitschaft des Druckers zur Daten-

annahme überprüft hat. Die dadurch bedingte Ausgabegeschwindigkeit genügt in der Regel. Gehen wir nun aber einmal davon aus, daß wir mehrere Programme schreiben und diese auf dem Drucker ausgeben wollen. Nun können wir das System nicht benutzen, solange der Drucker in Aktion ist. Wir müssen bis zum Ende der Druckausgabe warten, bevor wir einen weiteren Teil des Programms editieren oder assemblieren können.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 10.1 Print Buffer

1                                          PAGE     ,132
2                                          TITLE    Figure 10.1 Print Buffer
3       0000                       ABS0    SEGMENT AT 0
4       0020                               ORG      4*8H
5       0020  ????????             TIMER_INT        DD       ?          ; Hardware interrupt for timer
6       005C                               ORG      4*17H
7       005C  ????????             PRINT_INT        DD       ?          ; BIOS Print interrupt
8       0408                               ORG      408H
9       0408  ????                 PRINTER_BASE     DW       ?        ; Base address for printer adapter
10      040A                       ABS0    ENDS
11      0000                       CODE    SEGMENT
12      0100                               ORG      100H               ; .COM file
13                                         ASSUME   CS:CODE,DS:CODE,ES:CODE
14      0100  EB 09 90                     JMP      START
15
16      0103  ????????             PRINT_VECTOR     DD       ?          ; Store original INT 17H
17      0107  ????????             TIMER_VECTOR     DD       ?          ; Store original INT 9
18
19      010B                       START:
20      010B  2B C0                        SUB      AX,AX              ; Establish addressing
21      010D  8E C0                        MOV      ES,AX
22                                         ASSUME   ES:ABS0
23      010F  26: A1 005C R                MOV      AX,WORD PTR PRINT_INT   ; Get original vectors
24      0113  26: 8B 1E 005E R             MOV      BX,WORD PTR PRINT_INT+2
25      0118  26: 8B 0E 0020 R             MOV      CX,WORD PTR TIMER_INT
26      011D  26: 8B 16 0022 R             MOV      DX,WORD PTR TIMER_INT+2
27      0122  A3 0103 R                    MOV      WORD PTR PRINT_VECTOR,AX        ; Save in this segment
28      0125  89 1E 0105 R                 MOV      WORD PTR PRINT_VECTOR+2,BX
29      0129  89 0E 0107 R                 MOV      WORD PTR TIMER_VECTOR,CX
30      012D  89 16 0109 R                 MOV      WORD PTR TIMER_VECTOR+2,DX
31
32                                 ;----- No interrupts while setting new values
33
34      0131  FA                           CLI
35      0132  26: C7 06 005C R 0162 R      MOV      WORD PTR PRINT_INT,OFFSET PRINT_HANDLER
36      0139  26: 8C 0E 005E R             MOV      WORD PTR PRINT_INT+2,CS
37      013E  26: C7 06 0020 R 0196 R      MOV      WORD PTR TIMER_INT,OFFSET TIMER_HANDLER
38      0145  26: 8C 0E 0022 R             MOV      WORD PTR TIMER_INT+2,CS
39      014A  B0 36                        MOV      AL,00110110B
40      014C  E6 43                        OUT      43H,AL
41      014E  B0 00                        MOV      AL,0           ; Speed up the timer by 256
42      0150  E6 40                        OUT      40H,AL
43      0152  B0 01                        MOV      AL,1
44      0154  E6 40                        OUT      40H,AL
45      0156  FB                           STI
46      0157  8D 16 28FD R                 LEA      DX,BUFFER_END  ; Mark ending address
47      015B  CD 27                        INT      27H            ; Exit and stay resident
48
49      015D  00                   TIMER_COUNT      DB       0
50      015E  01ED R               BUFFER_HEAD      DW       BUFFER_START
51      0160  01ED R               BUFFER_TAIL      DW       BUFFER_START
52
53                                 ;----- This routine handles INT 17H BIOS calls
54
55      0162                       PRINT_HANDLER    PROC     FAR
56                                         ASSUME   CS:CODE,DS:NOTHING,ES:NOTHING
57      0162  0A E4                        OR       AH,AH
58      0164  74 05                        JZ       BUFFER_CHARACTER           ; Is this a char print
59      0166  2E: FF 2E 0103 R             JMP      PRINT_VECTOR               ; No, send to ROM
60      016B                       BUFFER_CHARACTER:
61      016B  FB                           STI
62      016C  53                           PUSH     BX
63      016D  51                           PUSH     CX
64      016E  56                           PUSH     SI
65      016F  2B C9                        SUB      CX,CX                      ; Time out counter
66      0171                       PRINT_LOOP:
67      0171  2E: 8B 1E 0160 R             MOV      BX,BUFFER_TAIL             ; Get end of buffer
68      0176  8B F3                        MOV      SI,BX
69      0178  E8 01E2 R                    CALL     ADVANCE_POINTER            ; Move the pointer to next
70      017B  2E: 3B 1E 015E R             CMP      BX,BUFFER_HEAD             ; Is there a place in the buffer
71      0180  74 0E                        JE       BUFFER_FULL                ; No, wait until there is
72      0182  2E: 88 04                    MOV      CS:[SI],AL                 ; Yes, store in buffer
73      0185  2E: 89 1E 0160 R             MOV      BUFFER_TAIL,BX             ; New end pointer
74      018A  B4 00                        MOV      AH,0                       ; Good return code from INT 17H
75      018C                       PRINT_RETURN:
76      018C  5E                           POP      SI
77      018D  59                           POP      CX
78      018E  5B                           POP      BX
79      018F  CF                           IRET
80      0190                       BUFFER_FULL:
```

```
 81    0190  E2 DF                       LOOP    PRINT_LOOP             ; Try again
 82    0192  B4 01                       MOV     AH,1                   ; Took too long, mark error
 83    0194  EB F6                       JMP     PRINT_RETURN
 84    0196                      PRINT_HANDLER   ENDP
 85
 86                              ;----- This routine gets control 4660 times a second
 87
 88    0196                      TIMER_HANDLER   PROC    FAR
 89                                      ASSUME  CS:CODE,DS:NOTHING,ES:NOTHING
 90    0196  50                          PUSH    AX
 91    0197  53                          PUSH    BX
 92    0198  2E: 8B 1E 015E R            MOV     BX,BUFFER_HEAD
 93    019D  2E: 3B 1E 0160 R            CMP     BX,BUFFER_TAIL         ; Anything in buffer.
 94    01A2  75 14                       JNZ     TEST_READY             ; Yes, move on
 95
 96                              ;----- This section handles the timer speedup
 97
 98    01A4                      TIMER_RETURN:
 99    01A4  5B                          POP     BX
100    01A5  2E: FE 06 015D R            INC     TIMER_COUNT            ; Increment the frequency divider
101    01AA  75 06                       JNZ     SKIP_NORMAL
102    01AC  58                          POP     AX                     ; This is the one in 256
103    01AD  2E: FF 2E 0107 R            JMP     TIMER_VECTOR           ;  Go to the ROM timer routine
104    01B2                      SKIP_NORMAL:
105    01B2  B0 20                       MOV     AL,20H                 ; This is the 255 out of 256
106    01B4  E6 20                       OUT     20H,AL                 ; Send EOI to interrupt controller
107    01B6  58                          POP     AX
108    01B7  CF                          IRET
109
110                              ;----- Character in buffer, try to print it
111
112    01B8                      TEST_READY:
113    01B8  52                          PUSH    DX
114    01B9  1E                          PUSH    DS
115    01BA  2B D2                       SUB     DX,DX
116    01BC  8E DA                       MOV     DS,DX                  ; Establish addressing
117                                      ASSUME  DS:ABS0
118    01BE  8B 16 0408 R                MOV     DX,PRINTER_BASE
119    01C2  42                          INC     DX                     ; Point to status port
120    01C3  EC                          IN      AL,DX
121    01C4  A8 80                       TEST    AL,80H                 ; Test for printer busy
122    01C6  74 16                       JZ      NO_PRINT
123    01C8  4A                          DEC     DX                     ; Point to data port
124    01C9  2E: 8A 07                   MOV     AL,CS:[BX]             ; Get the character to print
125    01CC  E8 01E2 R                   CALL    ADVANCE_POINTER
126    01CF  2E: 89 1E 015E R            MOV     BUFFER_HEAD,BX
127    01D4  EE                          OUT     DX,AL                  ; Send character
128    01D5  83 C2 02                    ADD     DX,2                   ; Control Port
129    01D8  B0 0D                       MOV     AL,0DH
130    01DA  EE                          OUT     DX,AL                  ; Strobe to printer
131    01DB  B0 0C                       MOV     AL,0CH
132    01DD  EE                          OUT     DX,AL
133    01DE                      NO_PRINT:
134    01DE  1F                          POP     DS
135    01DF  5A                          POP     DX
136    01E0  EB C2                       JMP     TIMER_RETURN           ; Exit through timer handler
137    01E2                      TIMER_HANDLER   ENDP
138
139    01E2                      ADVANCE_POINTER PROC    NEAR
140    01E2  43                          INC     BX                     ; Bump the pointer
141    01E3  81 FB 28FD R                CMP     BX,OFFSET BUFFER_END
142    01E7  75 03                       JNE     ADVANCE_RETURN         ; Test for end of buffer
143    01E9  BB 01ED R                   MOV     BX,OFFSET BUFFER_START ; Wrap around for buffer
144    01EC                      ADVANCE_RETURN:
145    01EC  C3                          RET
146    01ED                      ADVANCE_POINTER ENDP
147
148    01ED                      BUFFER_START    LABEL   BYTE
149    01ED  2710 [                      DB      10000 DUP(?)
150                    ??
151                          ]
152
153    28FD                      BUFFER_END      LABEL   BYTE
154    28FD                      CODE    ENDS
155
156                                      END
```

Abbildung 10.1 Druckpuffer

Unser Beispielprogramm kann dieses Problem verringern, jedoch nicht völlig aus der Welt schaffen. Wir stellen dazu einen Speicherbereich als Druckpuffer bereit. Dieser Bereich bleibt dauernd als Puffer reserviert. DOS zieht diesen Bereich vom gesamten, im System verfügbaren Speicher ab. Haben wir also ein System nit 96K Bytes und reservieren 10K als Druckpuffer, so können wir den Makro-Assembler nicht mehr verwenden. Der Makro-Assembler benötigt nämlich 96K, und nach Abzug unseres Druckpuffers bleiben im System nur noch 86K Bytes frei. Bevor wir den Druckpuffer also verwenden, müssen wir uns überzeugen, daß noch genügend Speicher im System frei bleibt.

Nun zur Arbeitsweise der gepufferten Druckausgabe. Dabei wird der PRINT-Befehl (INT 17H) durch eine Routine ersetzt, die die auszugebenden Zeichen in einem Puffer ablegt anstatt sie direkt an den Drucker zu senden. Dies nennen wir „Puffern" der Druckausgabe. Ein anderer Teil des Programms, die eigentliche Druckausgabe, nimmt dann die Zeichen aus dem Puffer und gibt sie an den Drucker weiter.

Wichtig ist hier das Ersetzen der ROM BIOS-Routine für INT 17H. Praktisch alle Programme verwenden diese BIOS-Funktion zur Ausgabe von Zeichen auf den Drukker. Das heißt, daß alle normalen Druckausgaben nun nicht mehr an den Drucker, sondern an die Pufferroutine erfolgen. Speziell für dieses Beispiel können wir die Assemblerliste mit dem TYPE-Kommando auf dem Bildschirm ausgeben und durch Drücken der Tasten CTL und PRTSCR diese Ausgabe auf den Drucker umleiten.

Senden wir eine Assemblerliste an den Drucker und das Pufferprogramm ist vorhanden, so wandern die Zeichen in den Pufferspeicher anstatt direkt zum Drucker zu gehen. Durch das Puffern wird die Druckausgabe nur unwesentlich langsamer. Haben wir die Datei an den Bildschirm geschickt (und damit in den Druckpuffer), so geht die Steuerung zurück an DOS. Wir können dann die eigentliche Druckausgabe durch erneutes Betätigen der Tasten CTL-PRTSCR unterbrechen. Die auszugebende Datei befindet sich im Druckpuffer, und DOS kann mit anderen Aufgaben, wie beispielsweise Editieren oder Assemblieren, fortfahren.

Nun zum zweiten Teil unseres Programms. Hier entnehmen wir die Zeichen aus dem Pufferbereich und senden sie an den Drucker. Diese Routine wird vom Zeitgeberinterrupt gesteuert. Nach jedem solchen Interrupt erhält die Druckroutine die Steuerung. Befindet sich nun ein Zeichen im Puffer und der Drucker ist empfangsbereit, so sendet die Routine dieses Zeichen an den Drucker. Auf diese Weise werden die Zeichen mit einer an den Drucker angepaßten Geschwindigkeit ausgegeben. Da die Druckroutine im „Hintergrund" abläuft, können wir im Vordergrund andere Aufgaben wie Editieren oder Assemblieren bewältigen.

Sehen wir uns nun das Programm in Abbildung 10.1 und die Art, in der die einzelnen Komponenten zusammenarbeiten, an. Zuerst definieren wir das Segment ABS0, das die vom Programm benötigten Interruptvektoren enthält. Wir ersetzen dabei die beiden Interrupts für Drucker (INT 17H) und Zeitgeber (INT 8). Beachten wir außerdem, daß ABS0 auch die Stelle PRINTER_BASE bestimmt. Diese Stelle enthält die Basisadresse für Drucker 0. Wir gehen nämlich davon aus, daß alle Druckausgaben auf den Systemdrucker erfolgen.

Das CODE-Segment ist der Teil des Programms, der speicherresident bleiben soll. Wir haben unser Programm als .COM-Datei mit ORG 100H erstellt. Das heißt, wir müssen die in Kapitel 5 beschriebenen Schritte zum Erzeugen einer .COM-Datei aus dem Linker-Output gehen. Im Programm verwenden wir die Adressen PRINT_VECTOR und TIMER_VECTOR, um die ursprünglichen Werte dieser Vektoren sicherzustellen. Obwohl wir nämlich die Werte ersetzen, müssen wir auch die Originalwerte für die Druckausgabe kennen.

Der erste Teil des CODE-Segments mit dem Anfang bei START dient der Initialisierung. Dabei werden die aktuellen Werte der beiden Interruptvektoren gelesen. Wir speichern diese Adressen in Datenbereichen des CODE-Segments. Wir ersetzen nun die Vektoren aus dem unteren Speicherbereich mit den neuen Werten für

unser Puffer- bzw. Druckprogramm. Beachten wir dabei den Befehl CLI, der vor dem Durchführen dieser Operation die Interrupts ausschaltet. Nachdem unser Programm auch den Zeitgeberinterrupt verändert, können wir in diesem Zeitraum keinen weiteren Zeitgeber-Tick zulassen. Würde nämlich ein solcher Interrupt auftreten während unser Programm erst eines der beiden Wörter des entsprechenden Interruptvektors geändert hat, dann könnte der Prozessor an einer völlig undefinierten Stelle mit der Verarbeitung fortfahren. Es ist also besser, die Interrupts auszuschalten, als darauf zu hoffen, daß das System schon nicht irgendwohin verzweigen werde.

Bevor wir nun die Interrupts wieder einschalten, modifizieren wir den Zählwert des Zeitgebers. Normalerweise tritt eine Zeitgeberunterbrechung etwa 18mal pro Sekunde auf. Der Drucker kann jedoch mit 80 Zeichen pro Sekunde drucken. Würde unser Druckprogramm nun mit jedem Zeitgeber-Tick ein Zeichen ausgeben, so läge die maximale Druckgeschwindigkeit bei 18 Zeichen pro Sekunde. Durch beschleunigen des Zeitgebers erhalten wir nun mehr Zeitgeber-Ticks. Damit kann unser Programm den Drucker mit der vollen Geschwindigkeit von 80 Zeichen pro Sekunde bedienen. Im Beispiel laden wir dazu den Zählwert 256 in den Zeitgeber. Dies ist 256mal weniger als der normale Wert. Die TIMER_HANDLER-Routine benötigt jedoch ihrerseits einen Teil des Zeitgewinns.

Die Initialisierungsroutine kehrt über INT 27H wieder zu DOS zurück. Vor Programmende wird noch schnell das DX-Register auf die Adresse des ersten freien Bytes nach Programmende gesetzt. Beachten Sie, daß beide Programme und der Druckpuffer sich innerhalb dieses Bereichs befinden. Entsprechend den Regeln für INT 27H wird DOS diesen Bereich fortan unangetastet lassen.

Wir verschwenden allerdings etwas Speicherplatz. Der Initialisierungsteil des Programms wird nämlich nur einmal ausgeführt. Es gibt also keinen Grund, weshalb er speicherresident bleiben sollte. Wir könnten das Programm optimieren, indem wir den Teil zwischen START und INT 27H hinter das Label BUFFER_END verschieben. In diesem Falle würde bei Ausführung von INT 27H der Initialisierungsteil des Programms außerhalb des geschützten Bereichs liegen. Das nächste von DOS geladene Programm (z.B. der Assembler) würde dann diesen Programmteil überlagern. Allerdings erscheint es nicht gerade notwendig, 90 Bytes bei einem Bedarf von mehr als 10000 einzusparen. Aber die Möglichkeit besteht, falls Sie sie verwenden wollen.

Nun zur Routine PRINT_HANDLER. Sobald ein Programm INT 17H für eine Druckausgabe anspricht, erhält diese Routine anstelle des ROM BIOS die Steuerung. Die ersten drei Befehle dienen der Steuerungsübernahme vom BIOS. Die Routine arbeitet nur wenn ein Zeichen ausgegeben werden soll, also AH=0 ist. Alle anderen Funktionen werden wieder vom normalen ROM BIOS übernommen. Dementsprechend müssen wir auf AH=0 überprüfen. Ist die Bedingung nicht erfüllt, führen wir einen indirekten Sprung auf die gesicherte Adresse des ursprünglichen Interruptvektors aus. Dadurch erhält die Routine des ROM BIOS wieder die Steuerung und führt die gewünschte Funktion aus. Das heißt, wir müssen unser Programm nur für genau die Funktionen schreiben, die wir tatsächlich ändern wollen.

Zwei Dinge müssen wir allerdings bei dieser Art der Druckausgabe beachten. Erstens ist es nicht sehr sinnvoll, alle Druckerfunktionen mit Ausnahme von AH=0

an das BIOS weiterzuleiten. Initialisiert nämlich ein Programm den Drucker (AH=2) während die Pufferroutine aktiviert ist, so übernimmt BIOS die Steuerung und sendet einen Resetbefehl an den Drucker. Dieser Befehl löscht im Drucker die gerade in der Ausgabe befindliche Zeile. In den meisten Fällen bedeutet dies den Verlust von einem oder mehreren Zeichen. Wollen wir unser Programm also etwas mehr narrensicher machen, dann müßten wir alle Funktionen der Druckerroutine selbst bedienen.

Das zweite, das wir beachten müssen, ist die Verwendung des sichergestellten Drucker-Interruptvektors. Wir könnten ganz einfach die BIOS-Liste im Technical Reference Manual verwenden und die Startadresse der Druckerroutine heraussuchen. Und wir könnten diese Adresse direkt in unser Programm einsetzen, so wie wir es mit anderen absoluten Adressen auch tun. Das würde unser Programm aber an diese ROM BIOS-Adresse binden. Falls IBM nun irgendeinmal die Routinen des ROM BIOS ändert und so auch die Adresse der Druckerroutine modifiziert, dann läuft unser Programm nicht mehr. Solange Sie dabei das Programm nur für Ihren eigenen Rechner schreiben, niemals einen neuen anschaffen und das Programm auch nicht anderweitig verkaufen wollen, wird auch nie ein Problem auftauchen. Doch ist es in jedem Fall besser, absolute Adressen zu vermeiden, soweit es eine Möglichkeit dazu gibt. Und in unserem Beispiel kann die Initialisierungsroutine den Drucker-Interruptvektor leicht dazu verwenden, die ROM-Adresse der BIOS-Druckerroutine festzustellen.

Der Rest der PRINT_HANDLER-Routine speichert die Zeichen im Druckpuffer. Dabei wird vor dem Ablegen des Zeichens im Puffer auf freien Platz überprüft. Ist der Puffer voll, wartet die Routine bis sich wieder freier Platz ergibt. Da auch die normale BIOS-Routine wartet bis der Drucker wieder zur Aufnahme eines weiteren Zeichens bereit ist, sollte diese Wartezeit keine Probleme verursachen. Aus Sicherheitsgründen wird im CX-Register gezählt, wie oft die Druckerroutine bei einem Zeichen nicht verfügbar war. Sind dabei 64K Durchläufe erreicht, und der Puffer ist immer noch voll, ist sicher etwas nicht in Ordnung. Ebenso wie die Routine des ROM BIOS gibt auch PRINT_HANDLER hier einen Time-Out Fehler zurück.

Zusätzlich verwenden wir in der Druckerroutine ein Unterprogramm ADVANCE_POINTER. Dieses Unterprogramm macht aus dem Druckpuffer einen Ringpuffer. Erreicht der Pufferzeiger dabei das Ende des Puffers, so wird er wieder auf Pufferanfang zurückgesetzt. Die Routine entspricht der Tastaturpufferroutine im ROM BIOS. Allerdings umfaßt der Puffer hier nicht 16 Bytes, sondern 10000.

Die interessanteste Aufgabe im Beispiel hat die Routine TIMER_HANDLER. Bei der Initialisierung wurde sie mit dem Hardware-Zeitgeberinterrupt verbunden. Sie erhält also mit jedem Zeitgeber-Tick die Steuerung. Sie muß neben dem Senden der Zeichen an den Drucker auch die normalen Aufgaben der Zeitgeberroutine wie Tageszeit etc. übernehmen und dabei die von uns gesetzte höhere Geschwindigkeit des Zeitgebers wieder kompensieren.

Als erstes wird überprüft, ob ein Zeichen auszugeben ist. Es ist nämlich sinnlos, Zeichen auf den Drucker ausgeben zu wollen, wenn keine vorhanden sind. Ist also nichts auszugeben, springt das Programm sofort zu TIMER_RETURN. Hier wird die höhere Geschwindigkeit des Zeitgebers ausgeglichen.

Das Label TIMER_RETURN zeigt schon an, daß hier die normalen Zeitgeberfunktionen behandelt werden. Bei jedem Zeitgeberinterrupt wird ein 1-Byte Wert, TIMER_ COUNT, erhöht. Ist dieses Byte nicht Null, so wird die Unterbrechungsbehandlung durch einen EOI-Befehl an den Interruptkontroller abgeschlossen. Ist das Byte jedoch Null, so wird die Routine durch einen indirekten Sprung auf die gesicherte Adresse TIMER_VECTOR verlassen. Dadurch geht die Steuerung zurück an die Routine des ROM BIOS, die Tageszeit und Diskettenmotor bedient. Wir müssen diese Operationen also nicht durch unser Programm ausführen lassen. Den Sprung ins ROM BIOS führen wir nur nach jedem 256.Durchlauf durch unsere eigene Zeitgeberroutine aus. Nachdem wir aber auch die Geschwindigkeit des Zeitgebers um das 256-fache erhöht haben, erhält die Routine des ROM BIOS nach wie vor 18.2mal pro Sekunde die Steuerung. Das heißt, die Tageszeit läuft weiterhin korrekt, und auch der Diskettenmotor wird rechtzeitig abgeschaltet. Dies ist auch der Grund, weshalb wir den Wert 256 als Erhöhungsfaktor für den Zeitgeber wählten, wo doch bereits eine Beschleunigung auf das Fünffache genügt hätte, den Drucker mit voller Geschwindigkeit laufen zu lassen.

Wir wählten den Faktor 256 aus Gründen der Einfachheit. Wäre die Leistungsfähigkeit unseres Programmes ausschlaggebend gewesen, dann hätten wir den Faktor 5 wählen müssen. Die Abarbeitung jedes Zeitgeberinterrupts dauert nämlich mindestens 10 Mikrosekunden, und noch länger, wenn der Druckpuffer Zeichen enthält. Und die Zeit, die zur Bearbeitung dieses Interrupts benötigt wird, geht von der Verfügbarkeit des Systems für die andere auszuführende Aufgabe, wie etwa den Assembler, ab. Bei einer solchen Häufigkeit von Zeitgeberunterbrechungen wird nämlich das Programm im Vordergrund bereits langsamer. Für eine optimale Performanz Ihres Systems sollten Sie deshalb einen Wert unter 256 zur Erhöhung der Zeitgebergeschwindigkeit wählen.

Was geschieht nun in der Zeitgeberroutine, wenn ein Zeichen ausgegeben werden soll? Zuerst wird der Statusport gelesen, um festzustellen, ob der Drucker ein Zeichen annehmen kann. Wir verwenden dazu die Basisadresse im BIOS-Datenbereich, so daß unsere Routine sowohl mit der Drucker- als auch mit der Schwarz/Weiß-Karte läuft. Ist der Drucker nicht bereit, geht die Routine weiter zu TIMER_RETURN, wo nötigenfalls die Zeitgeberaufgaben erfüllt werden. Die eigentliche Druckroutine wartet nämlich nicht, bis der Drucker bereit ist. Wir wissen ja, daß wir durch den Zeitgeberinterrupt sehr schnell wieder hierher kommen und dann wiederum eine Druckausgabe versuchen können. Ein Warten auf den Drucker würde an dieser Stelle das ganze System behindern. Und das Ergebnis wäre das gleiche wie ohne Pufferroutine.

Ist der Drucker bereit, so entnimmt die Routine ein Zeichen aus dem Puffer und sendet es an den Drucker. Auch hier tut unsere Routine nicht alles, was eigentlich nötig wäre. Das ROM BIOS prüft nämlich bei jedem Senden eines Zeichens an den Drukker auf Fehlerbedingungen. Wir sollten es genauso machen. Doch was geschieht, wenn nun wirklich ein Problem auftritt? Wenn die Druckroutine einen Druckerfehler entdeckt, wie kann sie ihn an das Programm übermitteln, das die Ausgabe erzeugte? In einigen Fällen wird das Programm, das die Zeichen ausgab, bereits beendet sein. Der beste Weg ist, die Zeitgeberroutine bei jeder Ausgabe eines Zeichens auf Fehler prüfen zu lassen. Tritt ein Fehler auf, so kann PRINT_HANDLER die-

sen bei der nächsten Zeichenausgabe über INT 17H dem aufrufenden Programm mitteilen. Dies ist sicherlich keine ideale Lösung, aber vermutlich immer noch der beste Weg.

Bevor wir unser Beispiel verlassen, sollten wir noch ein Problem bedenken. Es gibt auch noch andere Routinen, die die Zeitgeberrate verändern. Auch BASICA, eine Art fortgeschrittenes BASIC, verändert den Zeitgeber in ähnlicher Weise wie wir. Rufen wir also nach dem Initialisieren unserer Pufferroutine BASICA auf, so erhält TIMER_ HANDLER die Unterbrechungen nicht länger in der eigentlich nötigen Folge. Da TIMER_HANDLER den Übergang der Steuerung an das BIOS kontrolliert, läuft die Tageszeit 256mal zu langsam. Auch initialisiert BASICA seinerseits den Drucker, was, wie wir bereits bemerkt haben, zu Kollisionen mit der Druckausgabe führt. Die gepufferte Druckausgabe ist also nicht in allen Fällen möglich. Aber sie erlaubt es uns, die Verwendung von INT 27H zum Schaffen einer permanenten neuen Systemfunktion zu erläutern. Außerdem haben wir in unserem Beispiel gezeigt, wie man durch Verändern der BIOS-Vektoren neue Funktionen in ein bestehendes Programm einfügen kann.

Laden in den oberen Speicherbereich

Der DOS-Interrupt INT 27H ist die bevorzugte Methode, permanente treiberähnliche Routinen in das System einzufügen. Und es ist ein schöner Weg, ein Programm dauernd verfügbar zu halten. Wir können solche Programme sogar in eine AUTOEXEC.BAT-Datei einfügen, so daß sie automatisch geladen werden. Dieses automatische Laden kann wünschenswert sein, wenn wir ein besonderes Ein/Ausgabegerät am System angeschlossen haben. DOS lädt dann bei jedem Starten des Systems die Gerätetreiberroutine automatisch mit. Und falls Sie unsere selbstgebaute Druckroutine über alles schätzen, so können Sie auch diese solchermaßen immer in das System laden.

Allerdings muß der DOS-Ausgang über INT 27H nicht immer funktionieren. IBM bietet für seinen PC nämlich drei verschiedene Betriebssysteme an: DOS, mit dem wir arbeiten; CP/M-86 von Digital Research; und schließlich das UCSD p-System von SofTech Microsystems. Außerdem gibt es noch einige unabhängige Softwareentwickler, die ihre eigenen Betriebssysteme anbieten. Wollen wir nun eine Gerätetreiberroutine entwickeln, die auf all diesen Systemen korrekt läuft, so müssen wir eine andere Methode als die bei DOS verwendete benutzen.

Nehmen wir an, Sie haben einen besonderen Drucker, den Sie als Zubehör zum IBM PC verkaufen wollen. Da dieser Drucker verhältnismäßig billig sein soll, benötigt er mehr Steuerung durch das BIOS als der IBM-Drucker. Sie entwickeln also den Drukker, die nötige Anschlußelektronik und eine BIOS-Routine, die den Drucker schließlich betreiben soll. Bei Verwendung des INT 27H können Sie dieses Geräte nur an Leute verkaufen, die DOS auf ihrem IBM PC verwenden. Sie benötigen deshalb eine Methode zum Laden von Gerätetreibern, die mit allen Betriebssystemen funktioniert.

Eine Methode, die nicht nur mit DOS funktioniert, wird als „Laden in den oberen Speicherbereich" bezeichnet. Dabei wird das System direkt nach dem Einschalt-

Selbsttest übernommen. Wir können dies dadurch erreichen, daß wir unsere eigene Boot-Diskette zum Laden des Systems verwenden. Wir speichern unser Programm also auf einer speziellen Diskette ab, und legen diese Diskette beim Einschalten des Rechners in das Laufwerk. Der Systemlader lädt dann den Gerätetreiber von der Diskette in den obersten Bereich des Schreib/Lesespeichers, also an die höchste verfügbare Adresse. Wir können dann die Variable im BIOS-Datenbereich, die die Speichergröße enthält, entsprechend anpassen. Dazu verringern wir einfach den verfügbaren Speicher um die Größe des in den obersten Speicherbereich geladenen Programms. Starten wir jetzt das normale Betriebssystem, so läuft alles wie gewohnt. Alle IBM-Betriebssysteme verwenden nämlich die BIOS-Speichergröße, um das Ende des verfügbaren Speicherbereichs zu bestimmen. Diese Systeme werden also das Programm im obersten Speicherbereich unberührt lassen. So lange auch Ihr System sich an diese Regeln hält, können Sie diese Methode des Ladens von Gerätetreibern verwenden.

Wir wollen noch ein Beispiel bringen, das diese Vorgehensweise zeigt. In Abbildung 10.2 sehen Sie die Assemblerliste von zwei Routinen. Die erste Routine behandelt das Initialisieren und Laden des Gerätetreibers. Die zweite ist der Treiber selbst. Wie Sie noch sehen werden, ist es nämlich günstiger, ein solches Programm in zwei eigenständige Teile aufzugliedern.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 10.2(a) Boot routine for RAM Disk
1                                        PAGE      ,132
2                                        TITLE     Figure 10.2(a) Boot routine for RAM Disk
3          0000                  NEW_DISK          SEGMENT
4          0000                  DISK_BIOS         LABEL    FAR
5          0003                          ORG       3
6          0003                  OLD_VECTOR        LABEL    WORD
7          0003                  NEW_DISK          ENDS
8
9          0000                  ABSO    SEGMENT   AT 0
10         004C                          ORG       13H*4
11         004C                  DISK_VECTOR       LABEL    WORD
12         0410                          ORG       410H
13         0410                  EQUIPMENT         LABEL    WORD
14         0413                          ORG       413H
15         0413                  MEMORY_SIZE       LABEL    WORD
16
17         = 00A0                DISK_SIZE         EQU      160
18         7C00                          ORG       7C00H                   ; Load point for boot record
19         7C00                  BOOT_RECORD       LABEL    FAR
20         7C00                  ABSO    ENDS
21         0000                  CODE    SEGMENT
22                                       ASSUME    CS:CODE,DS:ABSO
23         7C00                          ORG       7C00H
24         7C00  8C C8                   MOV       AX,CS
25         7C02  8E D8                   MOV       DS,AX
26         7C04  8E C0                   MOV       ES,AX
27         7C06  BE 7C00 R               MOV       SI,OFFSET BOOT_RECORD
28         7C09  BF 7A00 R               MOV       DI,OFFSET BOOT_RECORD-200H ; Move everything down in memory
29         7C0C  B9 0200                 MOV       CX,512
30         7C0F  F3/ A4                  REP       MOVSB                   ; Move it all
31         7C11  E9 7A14 R               JMP       NEXT_LOCATION-200H
32         7C14                  NEXT_LOCATION:
33         7C14  83 06 0410 R 40         ADD       EQUIPMENT,40H           ; Increment number of drives
34         7C19  A1 0413 R               MOV       AX,MEMORY_SIZE
35         7C1C  2D 00A0                 SUB       AX,DISK_SIZE
36         7C1F  A3 0413 R               MOV       MEMORY_SIZE,AX          ; Remove disk space from memory
37         7C22  B1 06                   MOV       CL,6
38         7C24  D3 E0                   SHL       AX,CL                   ; Multiply by 1024/16
39         7C26  8E C0                   MOV       ES,AX                   ; Segment value for new disk
40         7C28  B8 0201                 MOV       AX,201H                 ; Read sector into this area
41         7C2B  BB 0000                 MOV       BX,0
42         7C2E  B9 0002                 MOV       CX,2
43         7C31  BA 0000                 MOV       DX,0
44         7C34  CD 13                   INT       13H                     ; Do the disk read into new area
45         7C36  72 1A                   JC        BOOT_ERROR
46                                       ASSUME    ES:NEW_DISK
47         7C38  A1 004C R               MOV       AX,DISK_VECTOR
48         7C3B  26: A3 0003 R           MOV       OLD_VECTOR,AX
49         7C3F  A1 004E R               MOV       AX,DISK_VECTOR+2
50         7C42  26: A3 0005 R           MOV       OLD_VECTOR+2,AX         ; Move the current pointer
51         7C46  C7 06 004C R 0000       MOV       DISK_VECTOR,0           ;  to NEW DISK
52         7C4C  8C 06 004E R            MOV       DISK_VECTOR+2,ES        ; Move pointer to new BIOS routine
53         7C50  EB 06                   JMP       SHORT REBOOT            ; Read in the next diskette
```

```
54      7C52                    BOOT_ERROR:
55      7C52  BE 7A8F R                 MOV     SI,OFFSET ERROR_MSG-200H        ; Print error message
56      7C55  E8 7C7D R                 CALL    PRINT_MSG
57      7C58                    REBOOT:
58      7C58  BE 7A9C R                 MOV     SI,OFFSET BOOT_MSG-200H         ; Print boot real OS message
59      7C5B  E8 7C7D R                 CALL    PRINT_MSG
60      7C5E                    WAIT_BOOT:
61      7C5E  B4 00                     MOV     AH,0
62      7C60  CD 16                     INT     16H                     ; Wait for keyboard input
63      7C62  3C 20                     CMP     AL,' '                  ; Must be space
64      7C64  75 F8                     JNE     WAIT_BOOT
65      7C66  B8 0201                   MOV     AX,201H
66      7C69  BB 7C00                   MOV     BX,7C00H
67      7C6C  B9 0001                   MOV     CX,1
68      7C6F  BA 0000                   MOV     DX,0
69      7C72  8E C2                     MOV     ES,DX                   ; Boot in the real OS
70      7C74  CD 13                     INT     13H
71      7C76  72 DA                     JC      BOOT_ERROR
72      7C78  EA 7C00 ---- R            JMP     BOOT_RECORD
73
74      7C7D                    PRINT_MSG       PROC    NEAR
75      7C7D  2E: 8A 04                 MOV     AL,CS:[SI]              ; Get the char to print
76      7C80  46                        INC     SI
77      7C81  3C 24                     CMP     AL,'$'                  ; End of message marker
78      7C83  75 01                     JNE     OUTPUT
79      7C85  C3                        RET
80      7C86                    OUTPUT:
81      7C86  B4 0E                     MOV     AH,14
82      7C88  BB 0000                   MOV     BX,0
83      7C8B  CD 10                     INT     10H                     ; Video BIOS print routine
84      7C8D  EB EE                     JMP     PRINT_MSG
85      7C8F  42 6F 6F 74 20 65 ERROR_MSG       DB      'Boot error',10,13,'$'
86            72 72 6F 72 0A 0D
87            24
88      7C9C  49 6E 73 65 72 74 BOOT_MSG        DB      'Insert new boot diskette',10,13
89            20 6E 65 77 20 62
90            6F 6F 74 20 64 69
91            73 6B 65 74 74 65
92            0A 0D
93      7CB6  48 69 74 20 73 70                 DB      'Hit space bar to continue',10,13,'$'
94            61 63 65 20 62 61
95            72 20 74 6F 20 63
96            6F 6E 74 69 6E 75
97            65 0A 0D 24
98      7CD2                    PRINT_MSG       ENDP
99      7CD2                    CODE    ENDS
100                                     END
```

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 10.2(b) RAM Disk Diver routine

1                                               PAGE    ,132
2                                               TITLE   Figure 10.2(b) RAM Disk Diver routine
3       0000                            CODE    SEGMENT
4                                               ASSUME  CS:CODE
5                                       ;-----------------------------------------
6                                       ;  This code becomes sector 1, track 0 of
7                                       ;   the RAM disk.  Any Reads/Writes to drive
8                                       ;   2 are directed here
9                                       ;-----------------------------------------
10      0000                            DISK    PROC    FAR
11      = 0140                          DISK_SIZE       EQU     320     ; Size of diskette in sectors
12      0000  EB 05 90                          JMP     START_BIOS
13      0003  ????????                  ORIGINAL_VECTOR DD      ?
14
15      0007                            START_BIOS:
16      0007  80 FA 02                          CMP     DL,2            ; Drive 2 Only
17      000A  74 05                             JE      L1
18      000C                            OLD_BIOS:
19      000C  2E: FF 2E 0003 R                  JMP     ORIGINAL_VECTOR ; Jump to ROM routine
20      0011                            L1:
21      0011  80 FC 01                          CMP     AH,1
22      0014  76 F6                             JBE     OLD_BIOS
23      0016  80 FC 04                          CMP     AH,4
24      0019  72 06                             JB      READ_WRITE      ; Handle read/write only
25      001B                            OK_RETURN:
26      001B  B4 00                             MOV     AH,0            ; OK return indicator
27      001D  F8                                CLC                     ; No error indicator
28      001E  CA 0002                           RET     2
29      0021                            READ_WRITE:
30      0021  53                                PUSH    BX
31      0022  51                                PUSH    CX
32      0023  52                                PUSH    DX
33      0024  56                                PUSH    SI
34      0025  57                                PUSH    DI              ; Save all registers
35      0026  1E                                PUSH    DS
36      0027  06                                PUSH    ES
37
38                                      ;----- Calculate transfer address
39
```

```
40    0028  50                     PUSH    AX              ; Save value
41    0029  B0 08                  MOV     AL,8            ; Sectors/track
42    002B  F6 E5                  MUL     CH
43    002D  B5 00                  MOV     CH,0
44    002F  03 C1                  ADD     AX,CX           ; Add in the sector number
45    0031  80 FE 00               CMP     DH,0            ; Head 0
46    0034  74 03                  JE      HEAD_0
47    0036  05 0140                ADD     AX,320          ; Move to second side
48    0039                HEAD_0:
49    0039  48                     DEC     AX              ; Adjust for zero origin
50    003A  3D 0140                CMP     AX,DISK_SIZE    ; Is it an OK value?
51    003D  76 0E                  JBE     DISK_OK
52    003F                RECORD_NOT_FOUND:
53    003F  58                     POP     AX
54    0040  07                     POP     ES
55    0041  1F                     POP     DS
56    0042  5F                     POP     DI              ; Recover all registers
57    0043  5E                     POP     SI
58    0044  5A                     POP     DX
59    0045  59                     POP     CX
60    0046  5B                     POP     BX
61    0047  B4 04                  MOV     AH,4            ; Record not found
62    0049  F9                     STC
63    004A  CA 0002                RET     2               ; Return with error
64    004D                DISK_OK:
65    004D  B1 05                  MOV     CL,5
66    004F  D3 E0                  SHL     AX,CL           ; Determine segment offset for disk
67    0051  8C C9                  MOV     CX,CS           ; Get base value of segment
68    0053  03 C8                  ADD     CX,AX           ; CX has disk segment
69
70    0055  51                     PUSH    CX
71    0056  8B D3                  MOV     DX,BX           ; Get transfer address
72    0058  B1 04                  MOV     CL,4
73    005A  D3 EA                  SHR     DX,CL           ; Segment for transfer address
74    005C  8C C1                  MOV     CX,ES
75    005E  03 D1                  ADD     DX,CX           ; DX has transfer segment
76    0060  59                     POP     CX
77    0061  81 E3 000F             AND     BX,0FH          ; Only low four bits in BX
78    0065  58                     POP     AX              ; Get back original parameters
79    0066  80 FC 02               CMP     AH,2
80    0069  74 11                  JE      READ_OPN
81    006B                WRITE_OPN:
82    006B  8C CE                  MOV     SI,CS
83    006D  3B CE                  CMP     CX,SI           ; Test for write over code
84    006F  74 1B                  JE      ALL_DONE
85    0071  8E C1                  MOV     ES,CX
86    0073  BF 0000                MOV     DI,0
87    0076  8E DA                  MOV     DS,DX
88    0078  8B F3                  MOV     SI,BX           ; Set transfer parms
89    007A  EB 09                  JMP     SHORT DO_MOVE
90    007C                READ_OPN:
91    007C  8E D9                  MOV     DS,CX
92    007E  BE 0000                MOV     SI,0
93    0081  8E C2                  MOV     ES,DX
94    0083  8B FB                  MOV     DI,BX
95    0085                DO_MOVE:
96    0085  8A E8                  MOV     CH,AL           ; Number of words/sector
97    0087  B1 00                  MOV     CL,0
98    0089  FC                     CLD
99    008A  F3/ A5                 REP     MOVSW           ; Do the move operation
100   008C                ALL_DONE:
101   008C  07                     POP     ES
102   008D  1F                     POP     DS
103   008E  5F                     POP     DI
104   008F  5E                     POP     SI              ; Recover all registers
105   0090  5A                     POP     DX
106   0091  59                     POP     CX
107   0092  5B                     POP     BX
108   0093  B4 00                  MOV     AH,0            ; OK status
109   0095  F8                     CLC
110   0096  CA 0002                RET     2
111   0099                DISK     ENDP
112   0099                CODE     ENDS
113                                END
```

Abbildung 10.2 (a) Boot-Routine für RAM Disk; (b) Treiberroutine für RAM Disk

Der Gerätetreiber in unserem Beispiel ist dabei die Speicherversion eines Diskettenlaufwerks, eine sogenannte RAM Disk. Wir nehmen dazu 160K Bytes Speicherbereich und verwenden ihn fortan als Diskette. Wir wählten 160K, da dies das kleinste von IBM unterstützte Diskettenformat ist. Natürlich können wir mit mehr Speicher auch eine größere Diskette simulieren. Wir können eine solche RAM Disk verwenden, um das Zeitverhalten von Programmen, die sehr häufig Diskettenzugriffe durchführen, erheblich zu verbessern. Verlagern wir beispielsweise den Assembler und das Assembler-Quellprogram auf eine solche simulierte Diskette, so läuft die Assemblierung in Sekunden anstelle von Minuten ab. Einige Programme

können dabei bis zu zehnmal schneller werden. Der Preis für eine derartige Verbesserung sind 160K Speicher, die wir für die Simulation der Diskette opfern müssen. Verfügen wir etwa über ein System mit 256K Speicher, und verwenden es hauptsächlich zum Editieren und Assemblieren, dann benötigen wir eigentlich nur 96K für den Assembler. Die restlichen 160K können wir als RAM Disk verwenden. Halten wir aber fest, daß der Inhalt der RAM Disk beim Abschalten des Gerätes verloren geht. Wir müssen also sicherstellen, daß vor dem Abschalten der Inhalt der RAM Disk auf eine echte Diskette ausgelagert wurde.

Das erste Programm in Abbildung 10.2 ist die Laderoutine. Sie befindet sich in Sektor 1 der Spur 0 der Boot-Diskette. Wir werden später noch erklären, wie wir das Programm dorthin bringen können. Die POST-Routine liest nach ihrem Ablauf den Inhalt von Sektor 1, Spur 0 an die Speicherstelle 0:7C00H und überträgt die Steuerung an diese Stelle. Dies ist übrigens auch der Weg, auf dem IBM DOS oder jedes andere Betriebssystem in den Speicher lädt. Alles, was wir dazu tun müssen, ist nur, unser eigenes kleines System zu laden.

Das Segment NEW_DISK definiert eine Adresse in der Treiberroutine, die wir als zweite Assemblerliste in Abbildung 10.2 sehen. Da die beiden Routinen getrennt assembliert werden, dient diese Segment-Deklaration zum Verknüpfen von Laderoutine und Gerätetreiber zur Ausführungszeit. Das Segment ABS0 bestimmt die Interruptvektoren, die die Laderoutine ersetzt. Und das CODE-Segment enthält das Programm, das dann von der Diskette geladen wird. Und dieses CODE-Segment ist der einzige Programmteil, der auf die Boot-Diskette geschrieben wird.

Das erste, was unser Programm nun tut, ist sich selbst an die Stelle 0:7A00H verschieben. Später in der Initialisierungsphase wird nämlich das eigentliche System geladen. Und dieses Laden geschieht an die Stelle 0:7C00H. Würde unsere Initialisierungsroutine sich also nicht selbst in Sicherheit bringen, so würde sie jetzt vom eigentlichen System überschrieben.

Bei der Adresse NEXT_LOCATION richtet die Initialisierungsroutine den Gerätetreiber ein. Dazu werden die Ausstattungsflags so modifiziert, daß sie ein Diskettenlaufwerk mehr anzeigen als mit den Schaltern auf der Systemplatine festgelegt ist. Damit täuschen wir dem Betriebssystem später vor, die RAM Disk sei ein Teil der vorhandenen Hardware. Als nächstes wird der Wert MEMORY_SIZE um 160K Bytes vermindert. Diesen Speicher benötigen wir zur Simulation der Diskette. Außerdem wollen wir damit verhindern, daß das Betriebssystem später versucht, diesen für die Diskettensimulation reservierten Speicher seinerseits zu verwenden. Die Routine ermittelt zudem die Segmentadresse für diesen 160K Bereich, da sie wissen muß, wohin der Gerätetreiber geladen werden soll. Nachdem die Adresse nun bekannt ist, liest die Initialisierungsroutine den Inhalt von Sektor 2 der Spur 0 der Boot-Diskette in den reservierten Bereich. Wir werden später erklären, wie man den Gerätetreiber in Sektor 2 ablegt, ebenso wie die Laderoutine in Sektor 1.

Nach dem Lesen und Übertragen des eigentlichen Gerätetreibers modifiziert die Initialisierungsroutine den Vektor für das Disketten-BIOS (INT 13H) so, daß er auf diese neue Treiberroutine zeigt. Wie im vorhergehenden Beispiel wird auch hier der alte Wert des Vektors sichergestellt. Die neue Treiberroutine benötigt nämlich diesen alten Vektor zum Lesen der echten Disketten im Unterschied zur simulierten

Diskette. Schließlich startet die Routine noch das eigentliche Betriebssystem. Sie teilt dem Benutzer mit, die Original-Systemdiskette einzulegen, wartet auf eine Bestätigung, und liest den Boot-Satz. (Hätte die Routine sich nicht vorher selbst in Sicherheit gebracht, würde sie jetzt zerstört werden.) Läuft alles ordnungsgemäß ab, so springt die Routine jetzt auf die erste Stelle des Boot-Satzes und gibt die Steuerung an das echte Betriebssystem ab.

Bevor wir weitergehen, sehen wir uns noch an, wie wir die neue Boot-Routine auf der Boot-Diskette ablegen können. Als erstes benötigen wir dazu eine leere, aber formatierte Diskette. Sie wird die neue Boot-Diskette. Wie die Kommandofolge in Abbildung 10.3 zeigt, wird die Routine ganz normal assembliert und gebunden. Wir verwenden nun das DOS DEBUG-Programm und laden die Boot-Routine. Sie wird von DEBUG an Offset 7C00H des von ihm benutzten Codesegments geladen. Wir setzen nun alle Register so, daß wir das ROM BIOS zum Schreiben eines einzelnen Diskettensektors verwenden können. Das 3-Byte Programm an der Adresse 200H erfüllt diese Aufgabe. Ist der Status nach Ausführung des Schreibbefehls in Ordnung, so wurde der Boot-Satz korrekt auf die Diskette geschrieben.

Um den Gerätetreiber nun auf Sektor 2 zu schreiben, führen Sie ganz einfach die nächsten Schritte in Abbildung 10.3 aus. Mit DEBUG-Befehlen laden wir die RAM Disk-Treiberroutine. Mit dem DEBUG Write-Befehl legen wir das Treiberprogramm in den relativen Sektor 1 (Sektor 2 von Spur 0) der Diskette in Laufwerk A: ab. Wir könnten diese Methode übrigens auch zum Schreiben des Boot-Satzes auf die Diskette verwenden.

In ähnlicher Weise wie zum Schreiben auf die Diskette können wir übrigens fast alle BIOS-Funktionen von DEBUG aus verwenden. Um festzustellen, was ein bestimmter BIOS-Aufruf tut, können wir ihn ganz einfach unter DEBUG ausführen. Dazu müssen wir nur die nötigen Register setzen und ein einfaches, drei Byte langes Programm schreiben, das einen Softwareinterrupt ausführt und wieder zu DEBUG zurückkehrt. Diese Methode ist auch sehr praktisch zum Testen Ihrer Gerätetreiberroutinen.

Nun wieder zurück zur RAM Disk-Treiberroutine im zweiten Teil von Abbildung 10.2. Diese Routine ist ein Gerätetreiber für eine simulierte Diskette. Halten wir fest, daß die Boot-Routine den ursprünglichen Interruptvektor für den Disketteninterrupt INT 13H bei Offset 3 des Segments sichergestellt hat. Die Treiberroutine benutzt diesen Vektor für alle Diskettenfunktionen, die nicht von der simulierten Diskette bearbeitet werden können. Wir gehen dabei davon aus, daß die RAM Disk auf Laufwerk 2 liegt. Bei jeder Anforderung für ein anderes Laufwerk geht die Routine weiter zum ROM BIOS und benützt dabei den in ORIGINAL_VECTOR sichergestellten Adresswert. Auch eine Anforderung zum Rücksetzen des Diskettenlaufwerks geht weiter an das ROM BIOS. Falls die angeforderte Funktion für das simulierte Laufwerk etwas anderes ist als ein Schreib- oder Lesebefehl, kehrt die Treiberroutine immer mit einem Status für korrekte Befehlsausführung zurück. Die RAM Disk benötigt übrigens kein Formatieren, und da es keine Fehlertests gibt, können wir auch keine Prüfbefehle ausführen.

Ist die angeforderte Funktion jedoch ein Schreib- oder Lesebefehl, so errechnet die Treiberroutine die Adresse des simulierten Diskettensektors im Speicher. Alles was

sich außerhalb der Diskette befindet, erzeugt einen Fehlercode „Satz nicht gefunden". Der Treiber setzt Quell- und Zielregister entsprechend der Richtung der auszuführenden Operation. Mit einem Befehl REP MOVSW werden schließlich die Daten zwischen der simulierten Diskette und dem Benutzerpuffer hin- und hertransportiert. Danach wird der Status korrekt gesetzt und zum aufrufenden Programm zurückgekehrt.

Das Beispiel zeigt, wie wir eine simulierte Diskette einrichten können. Allerdings ist das Beispiel nicht zum echten Einsatz geeignet. Um allgemein einsetzbar zu sein, müßten wir das Programm so modifizieren, daß es mit jedem simulierten Laufwerk und nicht nur mit Laufwerk 2 arbeiten kann. Auch könnten wir das Programm so verändern, daß es jede Sektorgröße verarbeiten kann, obwohl dies im allgemeinen nicht notwendig ist. Verwenden wir die Diskettensimulation ausschließlich mit DOS, so muß die Initialisierungsroutine die Diskette mit einem Inhaltsverzeichnis und einer Dateizuordnungstabelle (File Allocation Table — FAT) versehen. So wie die Routine jetzt aussieht, müssen wir nach dem Laden von DOS nämlich noch das Laufwerk C: formatieren. Dabei ist für die simulierte Diskette zwar kein physikalisches Formatieren nötig, doch erzeugt der FORMAT-Befehl von DOS eine FAT und ein Inhaltsverzeichnis, wie es für die weitere Verarbeitung unter DOS benötigt wird.

```
A>MASM BOOT,,,;
The IBM Personal Computer MACRO Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Warning Severe
Errors  Errors
0       0

A>B:LINK BOOT,,,;

IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982

 Warning: No STACK segment

There was 1 error detected.

A>MASM DISK,,,;
The IBM Personal Computer MACRO Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Warning Severe
Errors  Errors
0       0

A>B:LINK DISK,,,;

IBM Personal Computer Linker

Version 1.10 (C)Copyright IBM Corp 1982

 Warning: No STACK segment

There was 1 error detected.

A>B:DEBUG BOOT.EXE
-R

AX=0000  BX=0000  CX=7CD3  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=06D7  ES=06D7  SS=06E7  CS=06E7  IP=0000   NV UP DI PL NZ NA PO NC
06E7:0000 0000          ADD     [BX+SI],AL              DS:0000=CD
-U7C00 7C05
06E7:7C00 8CC8          MOV     AX,CS
06E7:7C02 8ED8          MOV     DS,AX
06E7:7C04 8EC0          MOV     ES,AX
-RAX
AX 0000
:301
-RBX
BX 0000
:7C00
-RCX
CX 7CD3
:1
-RDX
DX 0000
:
-RES
```

```
ES 06D7
:6E7
-E200
06D7:0200  00.CD   00.13   00.CC        ;*** Insert boot diskette here
-G=100

AX=0000  BX=7C00  CX=0001  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=06D7  ES=06E7  SS=06E7  CS=06E7  IP=0102   NV UP EI PL NZ NA PE NC
06E7:0102 CC            INT     3
-NDISK.EXE                              ;*** Insert program diskette here
--L
-U0 10
06E7:0000 EB05          JMPS    0007
06E7:0002 90            NOP
06E7:0003 0000          ADD     [BX+SI],AL
06E7:0005 0000          ADD     [BX+SI],AL
06E7:0007 80FA02        CMP     DL,02
06E7:000A 7405          JZ      0011
06E7:000C 2E            SEG     CS
06E7:000D FF2E0300      JMP     L,[0003]  ;*** Insert boot diskette here
-W0 0 1 1
-Q
A>
```

Abbildung 10.3 Schritte zum Laden in den oberen Speicherbereich

Außerdem speichert die Routine den Gerätetreiber in den simulierten Sektor 1 von Spur 0. DOS verwendet zwar diesen Sektor auf Laufwerk C: nicht, aber andere Systeme könnten dies tun. Wie Sie vielleicht bemerkt haben, verhindert die Simulation ein Schreiben auf diesen Sektor von Spur 0, zerstört also zumindest nicht sich selbst.

Allgemein gesagt ist die Technik des Ladens in den oberen Speicherbereich einigermaßen schwerfällig. Wir müssen von zwei getrennten Disketten laden, was vom Betreiber einen zusätzlichen Bedienungsaufwand erfordert. Der DOS Interrupt 27H ist da wesentlich besser geeignet, außer Sie verwenden gerade ein anderes System. Dann könnte allerdings diese Methode des Ladens der Treiberroutine die einzig mögliche sein.

Assembler-Unterprogramme

Neben residenten Gerätetreibern und eigenständigen Programmen wird Assembler oft für Unterprogramme in großen Programmkomplexen verwendet, die vorzugsweise in einer höheren Programmiersprache geschrieben sind. Diese höheren Programmiersprachen wie BASIC oder Pascal erlauben es Ihnen, schnell und einfach ein großes Programm zu erstellen. Allerdings kann es durchaus vorkommen, daß diese Sprachen Ihnen nicht alles erlauben, was Sie mit dem Rechner gerne tun würden. Dies trifft ganz besonders auf PCs zu, da ein gutes Anwendungsprogramm hier Nutzen aus allen Fähigkeiten der Maschine ziehen sollte. Eine höhere Programmiersprache läßt dies möglicherweise nicht zu. Entweder sind bestimmte Funktionen nicht verfügbar (wie der Aufruf von BIOS-Routinen), oder der durch die Programmiersprache bedingte Overhead macht das Programm unvertretbar langsam (z.B. das Lesen von Speicherstellen in BASIC mittels PEEK und POKE).

Glücklicherweise verfügen aber fast alle höheren Programmiersprachen über einen Mechanismus, der den Aufruf von Assembler-Unterprogrammen gestattet. Wir können also bestimmte Dinge schnell und effizient in Assembler erledigen, und dann für den Rest des Programms wieder in die höhere Programmiersprache zurückkehren. Wir werden in diesem Abschnitt noch etliche Beispiele bringen, die verschiedene Methoden zum Anschluß von Assemblerroutinen an Programme in höheren Sprachen zeigen.

Assemblerroutinen für BASIC

Als erstes schreiben wir ein Maschinenprogramm für den BASIC-Interpreter. Der BASIC-Interpreter ist vermutlich das bekannteste Werkzeug zum Schreiben von Programmen für den PC. Die Assemblerroutine, die wir in unser Programm einbauen wollen, ist gut über 100 Bytes lang. Ein Programm von solcher Länge kann nur noch schwer als Teil eines BASIC-Programms eingegeben werden, ein Weg, den wir noch später beschreiben werden.

Die Funktion, um die wir unser BASIC-Programm erweiter wollen, ist die Fähigkeit, den Inhalt eines Graphikbildschirms auf den Drucker auszugeben. Der IBM-Drucker verfügt nämlich über die Möglichkeit hierzu. Mit den Graphikbefehlen können wir dabei die einzelnen Nadeln des Druckers ansprechen, so wie wir auch im Graphikmodus der Farbkarte die einzelnen Pixels ansprechen können. In Abbildung 10.4 sind die für das Beispiel nötigen Graphikbefehle zusammengefaßt. Kurz gesagt ermöglicht der Drucker die Graphikfunktionen über eine Reihe von ESC-Sequenzen. Dabei senden wir anstelle eines ASCII-Zeichens einfach ESC (ASCII 27) an den Drucker. Die darauffolgenden Werte bestimmen dann die Aktion des Druckers und nicht irgendein Zeichen für die Ausgabe. Wie wir aus Abbildung 10.4 ersehen, gibt es Befehle zum Übertragen von Bitmustern an den Drucker, so daß tatsächlich jedes beliebige Punktmuster darstellbar ist.

Befehl	Aktion
ESC+"3"+n	Setzen Zeilenvorschub auf n/216
ESC+"K"+n1+n2+v1...vk (k = n1 + 256*n2)	Drucken Punkte v1...vk als 480 Punkte/Zeile

Abbildung 10.4 Graphikbefehle für den Drucker

Das Programm der Abbildung 10.5 verwendet die Graphikbefehle, um ein Abbild des 320x200 Graphikbildschirms auf den Drucker auszugeben. Jeder Bildschirmpunkt wird dabei an den Drucker übertragen. Ist der Bildpunkt in der Hintergrundfarbe dargestellt, erscheint nichts auf dem Drucker. Hat der Punkt dagegen eine der drei Vordergrundfarben, so wird ein schwarzer Punkt auf das Papier ausgegeben. Dabei werden die Daten allerdings nicht maßstäblich übertragen, so daß ein Kreis auf dem Bildschirm als Elipse auf dem Papier erscheint. Auch wird nicht zwischen den verschiedenen Vordergrundfarben unterschieden. Ein mehrfarbiges Bild ist also dann nur noch eine Mischung aus Schwarz und Weiß.

Die Routine PRINT_SCREEN ist eine FAR-Prozedur. Da ein Unterprogrammaufruf von BASIC ein FAR CALL ist, müssen wir auch unsere Routine entsprechend aufbauen. Die Zeichenfolge ESC+"3"+24 setzt den Zeilenvorschub des Druckers so, daß eine Punktreihe unmittelbar auf die andere folgt. Der Druckerkopf besteht aus acht Nadeln, die jeweils einen Abstand von 1/72 Zoll voneinander haben. Setzen wir den Zeilenvorschub also auf 8/72 Zoll (oder 24/216 Zoll), so erreichen wir, daß die ausgegebenen Zeilen direkt aneinander anschließen. Im Beispiel sehen Sie, wie die ESC-Sequenz an den Drucker übermittelt wird. Diese Folge von Zeichen und

Zahlenwerten wird an den Drucker ausgegeben, wie wenn es sich um eine ganz normale Zeichenfolge handeln würde. Den Rest erledigt der Drucker selbst.

Jede Bewegung des Druckkopfes über das Papier druckt acht Zeilen (eine für jede Nadel des Kopfes) quer über alle 320 Spalten. Beim Label NEXT_ROW senden wir die Sequenz ESC+"K"+64+1 an den Drucker. Dies gibt an, daß die nächsten 320 Bytes (64+1X256) Punktabbilder für den Graphikdrucker enthalten. Die Angabe „K" für den Graphikmodus gestattet uns die Ausgabe von maximal 480 Punkten quer über das Papier.

Wir verwenden nun die Video-BIOS Funktion, um die einzelnen Bildpunkte vom Bildschirm zu lesen. Dabei werden acht aufeinanderfolgende Zeilen der jeweils aktuellen Spalte gelesen und die Werte in einem einzigen Byte abgelegt. Eine „1" bedeutet dabei, daß es sich um eine Vordergrundfarbe handelt, die auf dem Drucker einen Punkt erzeugen soll. Das Programm läuft nun durch die Schleife NEXT_ COLUMN, bis alle 320 Spalten (was zugleich auch 320 Bytes bedeutet) an den Drucker übertragen sind. Dann gehen wir weiter zu den nächsten acht Zeilen, nachdem wir den Drucker durch Zeilenvorschub und Wagenrücklauf (ASCII 13 und 10) auf die nächste Zeile positioniert haben. Die 25 Zeilen auf dem Drucker decken dabei alle 200 Zeilen des Bildschirms ab. Mit einem FAR-Returnbefehl gehen wir dann zurück in den BASIC-Interpreter.

Das Unterprogramm PRINT nimmt unserem Programm einige Arbeit ab. Es sendet nämlich über die ROM BIOS-Funktion jeweils ein Byte an den Drucker. Dabei werden die nötigen Register für das ROM BIOS nur einmal gesetzt. Ohne das Unterprogramm müßten überall da, wo PRINT aufgerufen wird, die Register AH und DX immer wieder neu auf Null gesetzt werden.

```
The IBM Personal Computer MACRO Assembler 01-01-83              PAGE    1-1
Figure 10.5 Graphics Print Screen
1                                                   PAGE     ,132
2                                                   TITLE    Figure 10.5 Graphics Print Screen
3
4       = 001B                              ESC     EQU      27                  ; Escape ASCII Character
5
6       0000                                CODE    SEGMENT
7                                                   ASSUME   CS:CODE
8       0000                                PRINT_SCREEN     PROC      FAR
9       0000  B0 1B                                 MOV      AL,ESC              ; Set up 1/8" spacing
10      0002  E8 0060 R                             CALL     PRINT
11      0005  B0 33                                 MOV      AL,'3'
12      0007  E8 0060 R                             CALL     PRINT
13      000A  B0 18                                 MOV      AL,24               ;  as 24/216 "
14      000C  E8 0060 R                             CALL     PRINT
15      000F  BA 0000                               MOV      DX,0                ; Row number
16      0012                                NEXT_ROW:
17      0012  B0 1B                                 MOV      AL,ESC              ; Send out ESC K
18      0014  E8 0060 R                             CALL     PRINT
19      0017  B0 4B                                 MOV      AL,'K'
20      0019  E8 0060 R                             CALL     PRINT
21      001C  B0 40                                 MOV      AL,320-256          ; Byte count of 320
22      001E  E8 0060 R                             CALL     PRINT
23      0021  B0 01                                 MOV      AL,1
24      0023  E8 0060 R                             CALL     PRINT
25      0026  B9 0000                               MOV      CX,0                ; Column number
26      0029                                NEXT_COLUMN:
27      0029  52                                    PUSH     DX                  ; Save row number
28      002A  BB 0008                               MOV      BX,8                ; Number of bits to read
29      002D                                NEXT_DOT:
30      002D  D0 E7                                 SHL      BH,1                ; Open up least significant bit
31      002F  B4 0D                                 MOV      AH,13               ; Read dot
32      0031  CD 10                                 INT      10H                 ;  at current row and column
33      0033  0A C0                                 OR       AL,AL
34      0035  74 03                                 JZ       BACKGROUND          ; Test for background color
35      0037  80 CF 01                              OR       BH,1                ; Not background, turn on bit
36      003A                                BACKGROUND:
37      003A  42                                    INC      DX                  ; Move to next row
38      003B  FE CB                                 DEC      BL                  ; Decrement row count for this pass
39      003D  75 EE                                 JNZ      NEXT_DOT
```

```
40      003F  8A C7                   MOV    AL,BH        ; Dots to print
41      0041  E8 0060 R               CALL   PRINT        ; Send to printer
42      0044  5A                      POP    DX           ; Recover starting row of this pass
43      0045  41                      INC    CX           ; Move to next column
44      0046  81 F9 0140              CMP    CX,320       ; Last column?
45      004A  75 DD                   JNZ    NEXT_COLUMN
46      004C  B0 0D                   MOV    AL,13        ; Send carriage return and line feed
47      004E  E8 0060 R               CALL   PRINT
48      0051  B0 0A                   MOV    AL,10
49      0053  E8 0060 R               CALL   PRINT
50      0056  83 C2 08                ADD    DX,8         ; Move to next group of rows
51      0059  81 FA 00C8              CMP    DX,200       ; Have we done last row yet?
52      005D  72 B3                   JB     NEXT_ROW
53      005F  CB                      RET                 ; Return to BASIC
54      0060              PRINT_SCREEN       ENDP
55
56      0060              PRINT       PROC   NEAR
57      0060  52                      PUSH   DX
58      0061  B4 00                   MOV    AH,0         ; Print the character
59      0063  BA 0000                 MOV    DX,0         ;   in AL
60      0066  CD 17                   INT    17H
61      0068  5A                      POP    DX
62      0069  C3                      RET
63      006A              PRINT       ENDP
64      006A              CODE        ENDS
65                                    END
```

Abbildung 10.5 Druckausgabe des Graphikbildschirms

Wie kommen wir nun von BASIC in dieses Unterprogramm? Es gibt zwei Möglichkeiten, das Unterprogramm in ein BASIC-Programm einzuschließen. Läuft der BASIC-Interpreter, so verwendet er den restlichen freien Speicher bis zu einer maximalen Größe von 64K Bytes als Arbeitsbereich. Ist Ihr System größer als 96K, so gibt es bereits Bereiche, die BASIC nicht mehr erreichen kann. Das beste ist es nun, das Unterprogramm hier abzulegen. Verfügen Sie nicht über genügend Speicher, so können wir einen Teil des BASIC-Arbeitsbereichs reservieren, um dort unser Unterprogramm abzulegen. In einem späteren Beispiel werden wir noch eine Möglichkeit zeigen, das Unterprogramm im Variablenbereich von BASIC unterzubringen.

In Abbildung 10.6 sehen wir die einzelnen Schritte, die unser Unterprogramm für eine spätere Verwendung vorbereiten. Die nötigen Informationen sind in Anhang C des BASIC Reference Manual enthalten. Teil (a) ist dabei für eine Maschine mit einem Speicher von 96K oder mehr bestimmt. Wir assemblieren nun das Programm wie gewohnt. Beim Linken verwenden wir dann die /H-Angabe. Der Linker erstellt die .EXE-Datei nun so, daß das Programm an die höchste verfügbare Adresse im Speicher geladen wird, und nicht wie üblich an die niedrigste.

Um unser Unterprogramm nun mit dem BASIC-Programm zu verbinden, benötigen wir DEBUG. Wir starten DEBUG, laden BASIC, notieren uns die Registerinhalte und laden dann die Assemblerroutine. Unser Beispiel wurde übrigens auf einer Maschine mit 128K Speicher ausgeführt. Der Wert 1FF9H im CS-Register zeigt an, daß das Programm 70H Bytes vom Ende des Speichers geladen wurde. Nachdem das Programm 6AH Bytes lang ist, hat es der Linker unter Berücksichtigung der Paragraphengrenzen tatsächlich an die höchstmögliche Adresse abgelegt. Wichtig ist noch, daß das Programm über das Codesegment verschiebbar sein muß. Das heißt, wir können das Programm irgendwohin im Speicher verschieben, solange nur der erste Befehl den Offset 0 im aktuellen Codesegment besitzt. Wollen wir das Programm auf eine andere Maschine mit mehr oder auch weniger Speicher übertragen, kann diese Eigenschaft kritisch werden.

```
B>A:MASM FIG10-5,,,;
The IBM Personal Computer MACRO Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Warning Severe
Errors  Errors
0       0
```

```
B>A:LINK FIG10-5,,,/H;

IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982

 Warning: No STACK segment

There was 1 error detected.

B>A:DEBUG A:BASIC.COM
-R

AX=0000  BX=0000  CX=2B80  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=04B5  ES=04B5  SS=04B5  CS=04B5  IP=0100   NV UP DI PL NZ NA PO NC
04B5:0100 E91329        JMP     2A16
-NFIG10-5.EXE
-L
-R

AX=0000  BX=0000  CX=006A  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=04B5  ES=04B5  SS=1FF9  CS=1FF9  IP=0000   NV UP DI PL NZ NA PO NC
1FF9:0000 B01B          MOV     AL,1B

-RSS
SS 1FF9
:4B5

-RCS
CS 1FF9
:4B5

-RIP
IP 0000
:100

-G

----- In BASIC Interpreter, enter these commands

DEF SEG = &H1FF9
BSAVE "FIG10-5",0,&H70
```

(a)

```
B>A:MASM FIG10-5,,,;
The IBM Personal Computer MACRO Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Warning Severe
Errors  Errors
0       0

B>A:LINK FIG10-5,,,/H;

IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982

 Warning: No STACK segment

There was 1 error detected.

B>A:DEBUG A:BASIC.COM /M:&H8000
-R

AX=0000  BX=0000  CX=2B80  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=04B5  ES=04B5  SS=04B5  CS=04B5  IP=0100   NV UP DI PL NZ NA PO NC
04B5:0100 E91329        JMP     2A16
-NFIG10-5.EXE
-L
-R
AX=0000  BX=0000  CX=006A  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=04B5  ES=04B5  SS=0FF9  CS=0FF9  IP=0000   NV UP DI PL NZ NA PO NC
0FF9:0000 B01B          MOV     AL,1B

-RSS
SS 0FF9
:4B5

-RCS
CS 0FF9
:4B5

-RIP
IP 0000
:100

-G

----- In BASIC Interpreter, enter these commands

DEF SEG = &H0FF9
BSAVE "FIG10-5",0,&H70
```

(b)

Abbildung 10.6 (a) Erzeugen eines Unterprogramms für BASIC;
(b) Erzeugen eines Unterprogramms für BASIC auf 64K

Wir starten nun den BASIC-Interpreter. Dazu belegen wir die Register mit den Werten, die sie unmittelbar nach dem Laden von BASIC hatten. Nachdem BASIC läuft, verwenden wir den Befehl DEF SEG, um das Unterprogramm zu lokalisieren. Mit dem BSAVE-Befehl wird das Programm zurück auf die Diskette geschrieben und kann nun von BASIC mit dem BLOAD-Befehl wieder geladen werden.

In Teil (b) von Abbildung 10.6 wird das Vorgehen von Teil (a) wiederholt, nur daß es sich hier um einen Rechner mit 64K Speicher handelt. Der Unterschied liegt darin, daß BASIC nicht den ganzen Speicher als Arbeitsbereich verwenden kann. Die /M-Angabe im Befehl zum Starten von BASIC beschränkt den Arbeitsbereich und läßt so noch Platz für unser Unterprogramm. Wir werden einen ähnlichen Befehl dann auch noch benötigen, wenn wir das Programm laufen lassen.

Wir können unser Assemblerprogramm sogar mit DEBUG unter BASIC bearbeiten. Als Teil des G-Befehls, mit dem wir die Ausführung von BASIC starten, können wir nämlich einen Breakpoint in unserem Unterprogramm setzen. Wenn dieser nun erreicht wird, wird die Arbeit des BASIC-Interpreters unterbrochen und es erscheint wieder die gewohnte DEBUG-Registerausgabe.

Jetzt können wir das Assemblerprogramm als Teil eines normalen BASIC-Programms ausführen. Wir gehen wieder von einem 128K-System aus und geben die folgenden Befehle:

Eingabe „BASIC" auf der DOS-Befehlsebene

Eingabe „SCREEN1", sobald der BASIC-Interpreter läuft.

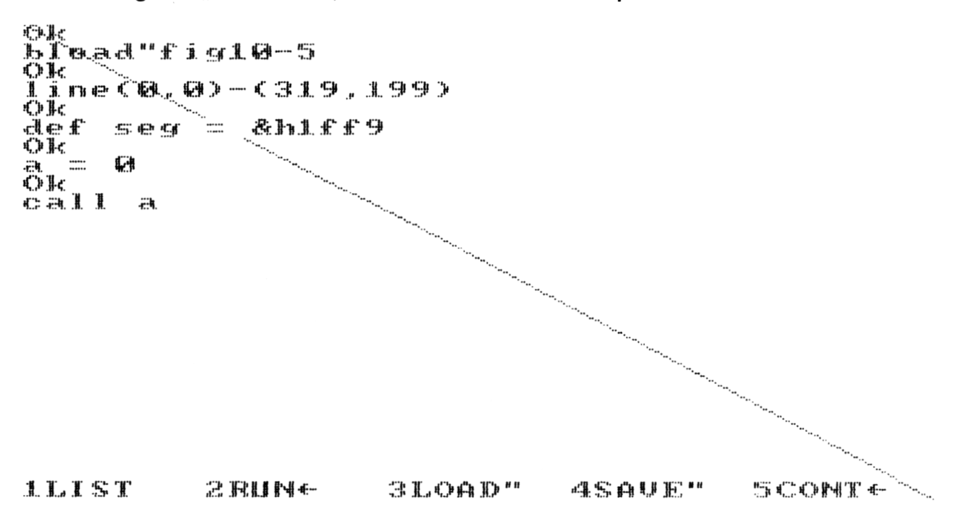


Abbildung 10.7 Druckausgabe vom Bildschirm

Damit befinden wir uns in BASIC und im 320x200 Graphikmodus. In Abbildung 10.7 sehen Sie dann den Rest der Geschichte. Mit dem BLOAD-Befehl laden wir das Unterprogramm an die gleiche Speicherstelle, aus der es vorher gesichert wurde. Bei diesem Befehl können wir aber auch Parameter verwenden, die das Programm

je nach Wunsch an eine andere Stelle im Speicher laden. Der Befehl LINE gibt der Graphik-Druckroutine schließlich etwas zu arbeiten. Zum Aufruf des Unterprogramms verwenden wir vorher DEF SEG, um das CS-Register richtig zu setzen. Der IP-Wert für die Routine paßt in eine einfache Variable. Der CALL-Befehl führt dann einen FAR-Aufruf an die gewünschte Adresse durch. In Abbildung 10.8 sehen Sie die Druckausgabe während des Ablaufs unseres Unterprogramms.

Verfügen wir nur über 64K Speicher, so würde sich das Beispiel in zwei Punkten vom vorhergehenden unterscheiden. Zum Starten des BASIC-Interpreters müßten wir den Befehl BASIC /M:&H8000 verwenden und damit den obersten Teil des Speichers für unser Assemblerprogramm reservieren. Und der Befehl DEF SEG würde schließlich auf die Adresse der Routine zeigen, wie wir sie in Abbildung 10.6 (b) gesehen haben.

Eingebaute Kurzprogramme

Im vorhergehenden Beispiel zeigten wir ein ziemlich großes Assemblerprogramm, das in einer eigenen Objektdatei abgelegt war und vom BASIC-Interpreter in den Speicher geladen wurde. Was tun wir aber, wenn das Programm recht klein ist? Es erscheint doch einigermaßen aufwendig, ein winziges Programm von einer eigenen Datei zu laden. In Anhang C des BASIC Reference Manual sehen wir eine Methode, ein Maschinenprogramm mit POKE in den Speicher außerhalb des BASIC-Arbeitsbereichs zu legen. Wir wollen aber in einem Beispiel noch einen anderen Weg zeigen.

```
The IBM Personal Computer MACRO Assembler 01-01-83           PAGE    1-1
Figure 10.8  Scroll routines for BASIC

1                                                PAGE    ,132
2                                                TITLE   Figure 10.8  Scroll routines for BASIC
3       0000                             CODE    SEGMENT
4                                                ASSUME  CS:CODE
5       0000                             SCROLL  PROC    FAR
6       0000  55                                 PUSH    BP
7       0001  8B EC                              MOV     BP,SP
8       0003  8B 76 06                           MOV     SI,[BP+6]       ; Get address of parm
9       0006  8B 0C                              MOV     CX,[SI]         ; Get the parm
10      0008  0A C9                              OR      CL,CL
11      000A  B7 07                              MOV     BH,7
12      000C  B8 0601                            MOV     AX,601H
13      000F  75 0C                              JNZ     WINDOW1         ; Determine which window
14      0011  B9 0200                            MOV     CX,200H         ; Set window 0
15      0014  BA 1010                            MOV     DX,1010H
16      0017                             DO_SCROLL:
17      0017  CD 10                              INT     10H
18      0019  5D                                 POP     BP
19      001A  CA 0002                            RET     2
20      001D                             WINDOW1:
21      001D  B9 0514                            MOV     CX,514H         ; Set window 1
22      0020  BA 1224                            MOV     DX,1224H
23      0023  EB F2                              JMP     DO_SCROLL
24      0025                             SCROLL  ENDP
25      0025                             CODE    ENDS
26                                               END
```

Abbildung 10.8 Scroll-Routinen für BASIC

In Abbildung 10.8 sehen Sie das Assemblerprogramm, das wir verwenden wollen. Es dient zum Aufrufen des ROM BIOS zum Verschieben des Bildschirms. Wie Sie an den Parametern im CX- und DX-Register sehen können, umfaßt das angegebene Fenster nur einen Teil des eigentlichen Bildschirms. Wir verwenden die Routine nun, um den Bildschirm in mehrere Fenster aufzuteilen, von denen jedes für sich verschoben werden kann. Da BASIC uns keine Möglichkeit dazu anbietet, benötigen wir hierfür ein Assemblerprogramm.

Wie Sie aus der Programmliste in Abbildung 10.8 sehen können, verwendet unser Programm einen Eingabeparameter, um festzulegen, welches der beiden Fenster verschoben werden soll. Das BASIC-Programm übergibt im CALL-Aufruf diesen Parameter an die Assemblerroutine. In Abbildung 10.9(a) sehen wir den Inhalt des Stacks, wenn BASIC das SCROLL-Unterprogramm aufruft. Der CALL-Befehl legt die Adresse jedes Arguments im Stack ab, bevor der FAR CALL auf das Unterprogramm ausgeführt wird. Die Adresse im Stack ist dabei der Offset des Arguments relativ zum DS-Register. Mit den ersten Befehlen der SCROLL-Routine wird diese Adresse nach SI geladen, wodurch der aktuelle Wert in das CX-Register geladen werden kann. In Abbildung 10.9(b) sehen wir den Inhalt des Stacks, nachdem die SCROLL-Routine den Inhalt des BP-Registers in diesen geschoben und dann BP mit SP geladen hat. Beachten wir, daß sich die Adresse des Arguments erst bei Byte 6 im Stack befindet. Hätte das BASIC-Programm mehrere Parameter übergeben, so wären diese in ähnlicher Weise vor dem CALL-Befehl in den Stack geschoben worden. Vorausblickend können wir gleich festhalten, daß bei Rückkehr aus dem Unterprogramm die Argumente mit einem Befehl RET 2 wieder aus dem Stack entfernt werden. BASIC erwartet nämlich von einem Unterprogramm, daß die übergebenen Parameter sich bei der Rückkehr nicht mehr im Stack befinden.

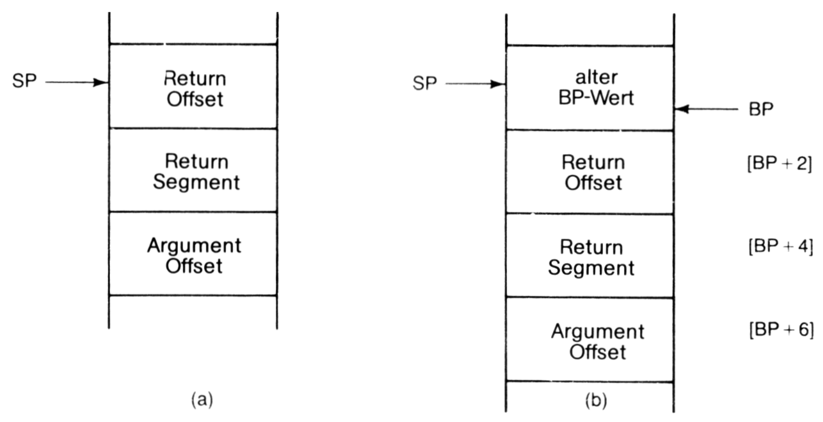


Abbildung 10.9 Stack für Unterprogrammaufruf

In Abhängigkeit vom übergebenen Parameter bearbeitet die SCROLL-Routine nun eines der beiden Bildschirmfenster. Ist der Wert 0, so wird der Inhalt des durch (2,0) und (16,16) bestimmten Fensters um eine Zeile nach oben verschoben. Ist der Wert nicht 0, so wird der Inhalt des Fensters von (5,20) und (18,36) um eine Zeile nach oben verschoben. Dabei wird nur der Text im definierten Fensterausschnitt bewegt, alle anderen Daten auf dem Bildschirm bleiben unverändert. Diese Fensterbearbeitung ist ein Teil der Scrollfunktion des BIOS. Um sie zu verwenden, müssen wir nur das BIOS mit den korrekten Parametern aufrufen.
In Abbildung 10.10 sehen wir das BASIC-Programm, das die SCROLL-Routine verwendet. Dieses einfache Beispiel schreibt nur einen Zeichenstring in jedes Fenster und ruft dann die SCROLL-Routine auf, um den Text zu verschieben. Das BASIC-Programm selbst erfüllt dabei keinen anderen Zweck als den Gebrauch der Fenster zu demonstrieren.

Worauf wir hier aber achten sollten, ist die Methode, mit der das Maschinenprogramm in das System geladen wird. Dabei enthält nämlich der String P$ das ganze Programm. Jedes Zeichen im String ist dabei ein Byte aus dem Objektcode von Abbildung 10.8. Dieses Programm wird dadurch eingegeben, daß wir einfach die Assemblerliste lesen und in das BASIC-Programm eintippen. Dies ist auch einer der Gründe, weshalb wir die Methode auf kurze Programme beschränken sollten. Es kann nämlich bei der Eingabe eines Programms auf diese Weise sehr leicht zu Fehlern kommen.

```
1 CLS
5 DEFINT A-Z
10 P$=CHR$(&H55)+CHR$(&H8B)+CHR$(&HEC)+CHR$(&H8B)+CHR$(&H76)+CHR$(&H6)
20 P$=P$+CHR$(&H8B)+CHR$(&HC)+CHR$(&HA)+CHR$(&HC9)+CHR$(&HB7)+CHR$(&H7)
30 P$=P$+CHR$(&HB8)+CHR$(&H1)+CHR$(&H6)+CHR$(&H75)+CHR$(&HC)+CHR$(&HB9)
40 P$=P$+CHR$(&H0)+CHR$(&H2)+CHR$(&HBA)+CHR$(&H10)+CHR$(&H10)+CHR$(&HCD)
50 P$=P$+CHR$(&H10)+CHR$(&H5D)+CHR$(&HCA)+CHR$(&H2)
60 P$=P$+CHR$(&H0)+CHR$(&HB9)+CHR$(&H14)
70 P$=P$+CHR$(&H5)+CHR$(&HBA)+CHR$(&H24)+CHR$(&H12)+CHR$(&HEB)+CHR$(&HF2)
100 ENTRY!=(PEEK(VARPTR(P$)+1))+(PEEK(VARPTR(P$)+2))*256
110 IF ENTRY!>32768! THEN ENTRY%=ENTRY!-65536! ELSE ENTRY%=ENTRY!
120 A$="ABCDEFGHIJ"
130 L=0:R=1
140 LOCATE 1,1:PRINT "Window Scrolling Example . . ."
200 LOCATE 15,1:PRINT A$;
210 CALL ENTRY%(L)
220 LOCATE 18,21:PRINT A$;
230 CALL ENTRY%(R)
240 A$=RIGHT$(A$,9)+ LEFT$(A$,1)
250 GOTO 200
```

Abbildung 10.10 BASIC-Programm für Bildschirmfenster

Nachdem nun das Maschinenprogramm in P$ enthalten ist, benutzt das BASIC-Programm die Funktion VARPTR zum Ermitteln der Adresse des Zeichenstrings. Da der CALL-Befehl nämlich die Adresse des Unterprogramms benötigt, müssen wir diese mit VARPTR ermitteln. Unter Verwendung der Information aus Anhang C des BASIC Reference Manual können wir die Adresse des Zeichenstrings aus dem zweiten und dritten Byte des Stringdeskriptors entnehmen. Und der von VARPTR zurückgegebene Wert ist die Adresse des Stringdeskriptors für P$. Wir entnehmen nun die Stringadresse aus dem Deskriptor und speichern sie in ENTRY!. Da dieser Wert sich zwischen 0 und 65,536 bewegen kann, müssen wir ihn in eine 1-Wort Integerzahl zwischen −32,768 und 32,767 umwandeln, die wir in ENTRY% ablegen. Mit dem Rest des Programms schreiben wir nur noch einen Zeichenstring in die Bildschirmfenster und rufen dann die SCROLL-Routine zum Verschieben des Textes auf.

Wenn Sie das Programm laufen lassen, werden Sie sehen, daß die Daten in den beiden Bildschirmfenstern unabhängig voneinander bewegt werden. Wir können also zwei Bildausschnitte definieren, und die Daten in diesen getrennt bewegen. In einem längeren Programm könnten wir sogar noch Linien um die Fenster ziehen und sie so vom restlichen Bildschirm abgrenzen. Die Verwendung dieser Fenstertechnik kann zu wirklich gutaussehenden Programmen führen, wenn Sie damit verschiedene Textbereiche darstellen wollen.

Bevor wir nun dieses Programm verlassen, wie können wir den maschinensprachlichen Teil auf Fehler prüfen? Zur Fehlersuche im Assemblerteil müssen wir das DOS DEBUG-Programm zur Hand haben. Dazu laden wir zuerst DEBUG und dann BASIC.COM (bzw. BASICA.COM, falls Ihr Programm Advanced BASIC benötigt). Nach dem Laden des eigentlichen BASIC-Programms ändern wir das erste Zeichen von P$ (und folglich das erste Byte der Assemblerroutine) auf CHR$(&HCC). Dies ist

der Wert für die Breakpoint-Unterbrechung INT 3. Läuft nun das BASIC-Programm ab und ruft die Assemblerroutine auf, so erhält DEBUG die Steuerung. Wir können dann den Wert 0CCH wieder durch den ursprünglichen Maschinencode (im Beispiel 055H) ersetzen. Dann können wir uns mit DEBUG durch das Assemblerprogramm arbeiten. Das wird in diesem Fall nicht lange dauern, da unsere Assemblerroutine recht kurz ist. Sie werden wahrscheinlich feststellen, daß die Probleme meist durch Schreibfehler beim Abtippen des Maschinencodes in das BASIC-Programm entstanden sind.

Kompilierte höhere Programmiersprachen

Im vorausgehenden Beispiel besprachen wir ein Assemblerprogramm in Verbindung mit dem BASIC-Interpreter. BASIC ist dabei als interpretierte Sprache Teil des IBM PC. Dies bedeutet, daß der Rechner das Programm in einer dem Original sehr ähnlichen Sprache speichert. Der Interpreter wandelt auch die BASIC-Befehle nicht in Maschinensprache um. Er sieht sich stattdessen während der Ausführung eines BASIC-Programms die einzelnen Befehle an und führt die dazu nötigen Schritte durch.

Ein Compiler arbeitet gänzlich anders. Er übersetzt die Befehle einer höheren Programmiersprache direkt in Maschinensprache. IBM bietet für den PC Compiler für BASIC, Pascal, FORTRAN und COBOL an. Die Ausgabe eines Compilers ist ein Programm in Maschinensprache (eine Datei mit Namen *.OBJ), das sehr der Ausgabe des Assemblers ähnelt. Das Erstellen eines Programms in einer höheren Programmiersprache ist also ein zweistufiger Vorgang. Zuerst muß das Programm übersetzt und gebunden werden. Erst dann können wir es ausführen. Ein interpretiertes Programm kann dagegen sofort ohne Übersetzung ausgeführt werden.

Die kompilierten höheren Programmiersprachen für den PC ähneln BASIC in der Tatsache, daß auch sie mit der Hardware nicht alles grundsätzlich Mögliche zulassen. Dabei läßt der BASIC-Interpreter sogar das Lesen und Schreiben der Ein/Ausgabeports bzw. von beliebigen Speicherstellen zu. Andere Programmiersprachen müssen nicht unbedingt über diese Fähigkeiten verfügen. Es kann also durchaus sein, daß Sie für Ihre FORTRAN- oder Pascal-Programme eher Assembler-Unterprogramme benötigen als bei BASIC. Sie müssen sie hier sogar verwenden, wenn Sie alle Möglichkeiten der Hardware ausschöpfen wollen.

Glücklicherweise ist die Verwendung von Unterprogrammen in Assembler bei einer kompilierten höheren Programmiersprache recht einfach. Die Ausgabe eines Compilers ist nämlich eine Objektdatei, die sofort gebunden werden kann. Auch die Ausgabe des Assemblers ist eine Objektdatei. Wir können deshalb mit dem DOS Linker das Assemblerprogramm und das Programm in der höheren Programmiersprache zusammenbinden. Wir müssen die Programme also nicht mehr erst während der Ausführungsphase zusammenbringen, wie wir es bei BASIC getan haben.

Führen wir dazu ein Beispiel mit FORTRAN durch. In Pascal läuft es übrigens ganz ähnlich. Unser Beispiel ähnelt dabei dem in Anhang D des FORTRAN Compiler Reference Manual. Dabei wird ein FORTRAN-Hauptprogramm mit einer Assembler-

routine verknüpft, die die Tageszeit über einen BIOS-Interrupt ermittelt. Die Assemblerroutine ruft BIOS auf, ermittelt die aktuelle Tageszeit und gibt diesen Wert an das FORTRAN-Programm zurück. Das Hauptprogramm wandelt dann den Zeitgeberwert in die Tageszeit in Stunden, Minuten und Sekunden um.

```
$STORAGE:4
        INTEGER A,HOURS,MINS,SECS,HSECS
        CALL TIMER(A)
        HOURS=A/65543
        A=A-HOURS*65543
        MINS=A/1092
        A=A-MINS*1092
        SECS=A/18
        HSECS=(100*(A-SECS*18))/18
        WRITE(*,10)HOURS,MINS,SECS,HSECS
10      FORMAT(1X,'THE TIME IS: ',I2,':',I2,':',I2,'.',I2)
        END
```

Abbildung 10.11 FORTRAN-Programm für Tageszeit

In Abbildung 10.11 sehen wir das FORTRAN-Hauptprogramm. Es ruft die externe Routine TIMER mit einem einzigen Parameter A auf. Diese Variable ist eine 4-Byte Integerzahl. Der Wert, den die Routine TIMER zurückgibt, ist die aktuelle Tageszeit, gemessen in Zeitgeber-Ticks seit Mitternacht. Das FORTRAN-Programm ermittelt nun aus diesem Wert die Zeit in Stunden (HOURS), Minuten (MINS), Sekunden (SECS) und Hundertstelsekunden (HSECS). Achten Sie dabei darauf, wie einfach die Multiplikation und Division von Zahlen in FORTRAN ist, und vergleichen Sie dies mit dem Aufwand, den wir für denselben Zweck in Assembler treiben mußten. Sie können hier sehr gut sehen, wie eine Verlagerung solcher Operationen in ein FORTRAN-Programm uns die Arbeit erleichtern kann. Besonders angenehm ist die Umwandlung von Integerzahlen in ausdruckbare Zeichen mit den FORTRAN-Anweisungen WRITE und FORMAT. Diese Aufgabe würde uns in einem Assemblerprogramm mehrere Hunderte von Befehlen kosten. Erinnern wir uns nur an das Beispiel mit dem 8087, wo ein Programm eine Gleitpunktzahl in einen ASCII-String umwandeln sollte. Dazu war eine ganze Reihe von Befehlen und die Verwendung des 8087 notwendig.

```
The IBM Personal Computer MACRO Assembler 01-01-83          PAGE    1-1
Figure 10.12  Assembly Routine for FORTRAN Main Program


1                                             PAGE     ,132
2                                             TITLE    Figure 10.12  Assembly Routine for FORTRAN Main Program
3                                     FRAME   STRUC
4        0000  ????                   SAVEBP  DW       ?
5        0002  ????????               SAVERET DD       ?
6        0006  ????????               A       DD       ?                ; Pointer to parameter
7        000A                         FRAME   ENDS
8
9        0000                         CODE    SEGMENT 'CODE'
10                                    DGROUP  GROUP    DATA
11                                            ASSUME   CS:CODE,DS:DGROUP,ES:DGROUP,SS:DGROUP
12       0000                         TIMER   PROC     FAR
13                                            PUBLIC   TIMER            ; Let linker know where it is
14       0000  55                             PUSH     BP
15       0001  8B EC                          MOV      BP,SP            ; Establish addressing to FRAME
16       0003  B4 00                          MOV      AH,0
17       0005  CD 1A                          INT      1AH              ; Call BIOS for Time of Day
18       0007  C4 5E 06                       LES      BX,[BP]+A        ; Get pointer in ES:BX to parameter
19       000A  26: 89 17                      MOV      ES:[BX],DX       ; Store low part of time
20       000D  26: 89 4F 02                   MOV      ES:[BX+2],CX     ; Store high part of time
21       0011  5D                             POP      BP
22       0012  CA 0004                        RET      4                ; Return and pop parm address
23       0015                         TIMER   ENDP
24       0015                         CODE    ENDS
25                                            END
```

Abbildung 10.12 Assemblerroutine für FORTRAN-Hauptprogramm

In Abbildung 10.12 finden wir das Unterprogramm TIMER. Sie sehen, daß diese einfache Routine einen BIOS-Interrupt verwendet, um die Tageszeit zu lesen. Sodann wird der Wert in einem Doppelwort gespeichert. Was wir dabei untersuchen wollen, ist die Art, in der die Parameter vom FORTRAN- an das Assemblerprogramm übergeben werden.

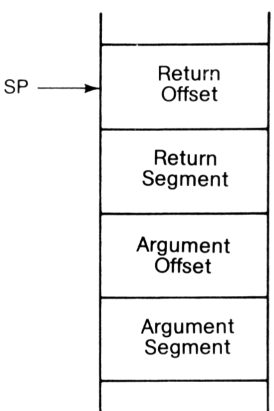


Abbildung 10.13 Stack für Unterprogrammaufruf von FORTRAN

In Abbildung 10.13 sehen wir den Stack zu Beginn des Ablaufs des Assemblerprogramms. So wie der BASIC-Interpreter übergibt auch das FORTRAN-Programm die Adresse der Parameter auf dem Stack. Allerdings übergeben der FORTRAN- bzw. Pascalcompiler einen Doppelwortpointer für die Parameter statt wie BASIC nur den Offset. Das bedeutet, daß unser Assemblerprogramm vor dem Zugriff auf die Parameter Segmentregister und Offsetadresse setzen muß. Sind mehrere Parameter vorhanden, so wären vom FORTRAN-Programm auch diese Adresswerte vor dem Unterprogrammaufruf im Stack abgelegt worden.

Das TIMER-Programm in Abbildung 10.12 adressiert den Stack, indem das BP-Register in diesen geschoben und sodann als neue Stackspitze verwendet wird. Die Struktur FRAME hilft uns, den Offset der einzelnen Parameter im Stack zu ermitteln, nachdem wir das BP-Register dort abgelegt haben. Mit LES BX,[BP]+A übernehmen wir die Adresse des Parameters in das Registerpaar ES:BX. Nun speichern wir nur noch die 4-Byte Tageszeit in die 4-Byte Integervariable an dieser Adresse.

Halten wir fest, daß die TIMER-Routine die Parameter als Teil des Returnbefehls aus dem Stack entfernt, so wie es auch BASIC-Programme tun. Beachten Sie auch, daß die Routine die PUBLIC-Anweisung verwendet, um das Label TIMER nach außen zu identifizieren. So kann der Linker die Routine finden und korrekt in das FORTRAN-Programm einbinden. Beim BASIC-Interpreter war das nicht nötig, da BASIC-Programm und Assemblerroutine nie zusammengebunden wurden.

Zusammenfassung

Assemblerprogrammierung ist ein mächtiges Werkzeug. Sie gestattet dem Programmierer völlige Kontrolle über die Funktionen der Hardware. Allerdings kann diese totale Kontrolle dazu führen, daß wir uns mit Details beschäftigen müssen, die mit dem eigentlichen Zweck unseres Programms kaum mehr etwas zu tun haben. Manchmal verkehrt sich die Leistungsfähigkeit der Assemblersprache eben auch in stumpfsinnige Detailbearbeitung.

In diesem letzten Kapitel zeigten wir noch, wie Sie die Leistungsfähigkeit der Assemblersprache gegen den Komfort einer höheren Programmiersprache abwägen können. Durch geeignete Aufteilung der zu erfüllenden Aufgaben kann ein geschickter Programmierer die unendliche Menge von Detailarbeit von der höheren Programmiersprache bewältigen lassen und sich selbst auf die eigentliche Problemlösung konzentrieren. Wo jedoch erhöhte Leistungsfähigkeit oder detaillierte Steuerung des Rechners nötig ist, kann er sich der Assemblersprache bedienen. Der Assembler gestattet es uns dann, Dinge zu tun, die in einer höheren Programmiersprache entweder überhaupt nicht möglich sind, oder aber in Anbetracht des durch diese Sprache bedingten Overheads zu lange dauern würden.

Es gibt zwei Wege, diese Arbeitsteilung zwischen Assembler und höheren Programmiersprachen durchzuführen. Die eine Möglichkeit ist, neue Treiberroutinen zu installieren, die dann standardmäßigen Zugriff auch auf völlig neue Geräte erlauben. Wir führten mehrere Beispiele für diese Vorgehensweise an, wie den gepufferten Druck und die Speicherversion der Diskette. Die andere Möglichkeit ist das Einfügen von Assemblerroutinen in größere Programme, wobei dann für bestimmte Zwecke jeweils ein bestimmtes Assemblerprogramm verwendet wird. Beide Wege sind gangbar, und in diesem letzten Kapitel zeigten wir noch einmal kurz alle bisher besprochenen Möglichkeiten, die Ihnen die Programmierung in Assembler gestattet.

Anhang A Befehlssatz des 8088

DATA TRANSFER

MOV = **Move**:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w 1
Immediate to register	1 0 1 1 w reg	data	data if w 1	
Memory to accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		

PUSH = **Push**:		
Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m
Register	0 1 0 1 0 reg	
Segment register	0 0 0 reg 1 1 0	

POP = **Pop**:		
Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m
Register	0 1 0 1 1 reg	
Segment register	0 0 0 reg 1 1 1	

XCHG = **Exchange**:		
Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m
Register with accumulator	1 0 0 1 0 reg	

IN=Input from:		
Fixed port	1 1 1 0 0 1 0 w	port
Variable port	1 1 1 0 1 1 0 w	

OUT = Output to:		
Fixed port	1 1 1 0 0 1 1 w	port
Variable port	1 1 1 0 1 1 1 w	
XLAT=Translate byte to AL	1 1 0 1 0 1 1 1	
LEA=Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m
LDS=Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m
LES=Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m
LANF=Load AH with flags	1 0 0 1 1 1 1 1	
SANF=Store AH into flags	1 0 0 1 1 1 1 0	
PUSHF=Push flags	1 0 0 1 1 1 0 0	
POPF=Pop flags	1 0 0 1 1 1 0 1	

ARITHMETIC

ADD = **Add**:				
Reg./memory with register to either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s w 01
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w 1	

ADC = **Add with carry**:				
Reg./memory with register to either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s w 01
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w 1	

INC = **Increment**:		
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m
Register	0 1 0 0 0 reg	
AAA=ASCII adjust for add	0 0 1 1 0 1 1 1	
BAA=Decimal adjust for add	0 0 1 0 0 1 1 1	

SUB = **Subtract**:				
Reg./memory and register to either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s w=01
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w 1	

SBB = **Subtract with borrow**				
Reg./memory and register to either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s w=01
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1	

DEC **Decrement**:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		

CMP **Compare**:				
Register/memory and register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w 01
Immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w 1	
AAS ASCII adjust for subtract	0 0 1 1 1 1 1 1			
DAS Decimal adjust for subtract	0 0 1 0 1 1 1 1			
MUL Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV Divide (unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV Integer divide (signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW Convert byte to word	1 0 0 1 1 0 0 0			
CWD Convert word to double word	1 0 0 1 1 0 0 1			

LOGIC

Instruction		
NOT Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m
SHL/SAL Shift logical/arithmetic left	1 1 0 1 0 0 v w	mod 1 0 0 r/m
SHR Shift logical right	1 1 0 1 0 0 v w	mod 1 0 1 r/m
SAR Shift arithmetic right	1 1 0 1 0 0 v w	mod 1 1 1 r/m
ROL Rotate left	1 1 0 1 0 0 v w	mod 0 0 0 r/m
ROR Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m
RCL Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m
RCR Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m

AND **And**:				
Reg./memory and register to either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w 1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w 1	

TEST **And function to flags, no result**:				
Register/memory and register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w 1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data	data if w 1	

OR **Or**:				
Reg./memory and register to either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w 1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w 1	

XOR **Exclusive or**:				
Reg./memory and register to either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w 1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w 1	

STRING MANIPULATION

Instruction	
REP=Repeat	1 1 1 1 0 0 1 z
MOVS=Move byte/word	1 0 1 0 0 1 0 w
CMPS=Compare byte/word	1 0 1 0 0 1 1 w
SCAS=Scan byte/word	1 0 1 0 1 1 1 w
LODS=Load byte/wd to AL/AX	1 0 1 0 1 1 0 w
STOS=Stor byte/wd from AL/A	1 0 1 0 1 0 1 w

CONTROL TRANSFER

CALL = Call:

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-high
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	
Direct intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	

JMP = Unconditional Jump:

Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within segment-short	1 1 1 0 1 0 1 1	disp	
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	

RET = Return from CALL:

Within segment	1 1 0 0 0 0 1 1		
Within seg adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ = Jump on equal/zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE = Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG = Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE = Jump on below/not above or equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA = Jump on below or equal/not above	0 1 1 1 0 1 1 0	disp	
JP/JPE = Jump on parity/parity even	0 1 1 1 1 0 1 0	disp	
JO = Jump on overflow	0 1 1 1 0 0 0 0	disp	
JS = Jump on sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ = Jump on not equal/not zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE = Jump on not less/greater or equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG = Jump on not less or equal/greater	0 1 1 1 1 1 1 1	disp	

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNB/JAE = Jump on not below/above or equal	0 1 1 1 0 0 1 1	disp
JNBE/JA = Jump on not below or equal/above	0 1 1 1 0 1 1 1	disp
JNP/JPO = Jump on not par/par odd	0 1 1 1 1 0 1 1	disp
JNO = Jump on not overflow	0 1 1 1 0 0 0 1	disp
JNS = Jump on not sign	0 1 1 1 1 0 0 1	disp
LOOP = Loop CX times	1 1 1 0 0 0 1 0	disp
LOOPZ/LOOPE = Loop while zero/equal	1 1 1 0 0 0 0 1	disp
LOOPNZ/LOOPNE = Loop while not zero/equal	1 1 1 0 0 0 0 0	disp
JCXZ = Jump on CX zero	1 1 1 0 0 0 1 1	disp

INT = Interrupt

Type specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
INTO = Interrupt on overflow	1 1 0 0 1 1 1 0	
IRET = Interrupt return	1 1 0 0 1 1 1 1	

PROCESSOR CONTROL

CLC = Clear carry	1 1 1 1 1 0 0 0	
CMC = Complement carry	1 1 1 1 0 1 0 1	
STC = Set carry	1 1 1 1 1 0 0 1	
CLD = Clear direction	1 1 1 1 1 1 0 0	
STD = Set direction	1 1 1 1 1 1 0 1	
CLI = Clear interrupt	1 1 1 1 1 0 1 0	
STI = Set interrupt	1 1 1 1 1 0 1 1	
HLT = Halt	1 1 1 1 0 1 0 0	
WAIT = Wait	1 0 0 1 1 0 1 1	
ESC = Escape (to external device)	1 1 0 1 1 x x x	mod x x x r/m
LOCK = Bus lock prefix	1 1 1 1 0 0 0 0	

Footnotes:

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low

if s:w = 01 then 16 bits of immediate data form the operand
if s:w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with ZF FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

Anhang B Befehlssatz des 8087

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Data Transfer				
FLD = LOAD				
Integer/Real Memory to ST(0)	ESCAPE MF 1	MOD 0 0 0 R/M	(DISP-LO)	(DISP-HI)
Long Integer Memory to ST(0)	ESCAPE 1 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
Temporary Real Memory to ST(0)	ESCAPE 0 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
BCD Memory to ST(0)	ESCAPE 1 1 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 1	1 1 0 0 0 ST(i)		
FST = STORE				
ST(0) to Integer/Real Memory	ESCAPE MF 1	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0) to ST(i)	ESCAPE 1 0 1	1 1 0 1 0 ST(i)		
FSTP = STORE AND POP				
ST(0) to Integer/Real Memory	ESCAPE MF 1	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to Long Integer Memory	ESCAPE 1 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to Temporary Real Memory	ESCAPE 0 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to BCD Memory	ESCAPE 1 1 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0) to ST(i)	ESCAPE 1 0 1	1 1 0 1 1 ST(i)		
FXCH = Exchange ST(i) and ST(0)	ESCAPE 0 0 1	1 1 0 0 1 ST(i)		
Comparison				
FCOM = Compare				
Integer/Real Memory to ST(0)	ESCAPE MF 0	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 0	1 1 0 1 0 ST(i)		
FCOMP = Compare and Pop				
Integer/Real Memory to ST(0)	ESCAPE MF 0	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 0	1 1 0 1 1 ST(i)		
FCOMPP = Compare ST(1) to ST(0) and Pop Twice	ESCAPE 1 1 0	1 1 0 1 1 0 0 1		
FTST = Test ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 0		
FXAM = Examine ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 1		

Arithmetic

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
FADD = Addition				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 0 0 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) and ST(0)	ESCAPE d P 0	1 1 0 0 0 ST(i)		
FSUB = Subtraction				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 1 0 R R/M	(DISP-LO)	(DISP-HI)
ST( ) and ST(0)	ESCAPE d P 0	1 1 1 0 R R/M		
FMUL = Multiplication				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 0 0 1 R/M	(DISP-LO)	(DISP-HI)
ST(i) and ST(0)	ESCAPE d P 0	1 1 0 0 1 R/M		
FDIV = Division				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 1 1 R R/M	(DISP-LO)	(DISP-HI)
ST(i) and ST(0)	ESCAPE d P 0	1 1 1 1 R R/M		
FSQRT = Square Root of ST(0)	ESCAPE 0 0 1	1 1 1 1 1 0 1 0		
FSCALE = Scale ST(0) by ST(1)	ESCAPE 0 0 1	1 1 1 1 1 1 0 1		
FPREM = Partial Remainder of ST(0) ÷ ST(1)	ESCAPE 0 0 1	1 1 1 1 1 0 0 0		
FRNDINT = Round ST(0) to Integer	ESCAPE 0 0 1	1 1 1 1 1 1 0 0		
FXTRACT = Extract Components of ST(0)	ESCAPE 0 0 1	1 1 1 1 0 1 0 0		
FABS = Absolute Value of ST(0)	ESCAPE 0 0 1	1 1 1 0 0 C 0 1		
FCHS = Change Sign of ST(0)	ESCAPE 0 0 1	1 1 1 0 0 0 0 0		

Transcendental

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
FPTAN = Partial Tangent of ST(0)	ESCAPE 0 0 1	1 1 1 1 0 0 1 0
FPATAN = Partial Arctangent of ST(0) ÷ ST(1)	ESCAPE 0 0 1	1 1 1 1 0 0 1 1
F2XM1 = $2^{ST(0)}-1$	ESCAPE 0 0 1	1 1 1 1 0 0 0 0
FYL2X = ST(1) · Log_2 [ST(0)]	ESCAPE 0 0 1	1 1 1 1 0 0 0 1
FYL2XP1 = ST(1) · Log_2 [ST(0) + 1]	ESCAPE 0 0 1	1 1 1 1 1 0 0 1

Constants

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
FLDZ = LOAD + 0.0 into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 1 1 0
FLD1 = LOAD + 1.0 into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 0 0
FLDPI = LOAD π into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 1 1
FLDL2T = LOAD $\log_2 10$ into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 0 1
FLDL2E = LOAD $\log_2 e$ into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 1 0
FLDLG2 = LOAD $\log_{10} 2$ into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 1 0 0
FLDLN2 = LOAD $\log_e 2$ into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 1 0 1

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Processor Control				
FINIT = Initialize NDP	ESCAPE 0 1 1	1 1 1 0 0 0 1 1		
FENI = Enable Interrupts	ESCAPE 0 1 1	1 1 1 0 0 0 0 0		
FDISI = Disable Interrupts	ESCAPE 0 1 1	1 1 1 0 0 0 0 1		
FLDCW = Load Control Word	ESCAPE 0 0 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
FSTCW = Store Control Word	ESCAPE 0 0 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
FSTSW = Store Status Word	ESCAPE 1 0 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
FCLEX = Clear Exceptions	ESCAPE 0 1 1	1 1 1 0 0 0 1 0		
FSTENV = Store Environment	ESCAPE 0 0 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
FLDENV = Load Environment	ESCAPE 0 0 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
FSAVE = Save State	ESCAPE 1 0 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
FRSTOR = Restore State	ESCAPE 1 0 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
FINCSTP = Increment Stack Pointer	ESCAPE 0 0 1	1 1 1 1 0 1 1 1		
FDECSTP = Decrement Stack Pointer	ESCAPE 0 0 1	1 1 1 1 0 1 1 0		
FFREE = Free ST(i)	ESCAPE 1 0 1	1 1 0 0 0 ST(i)		
FNOP = No Operation	ESCAPE 0 0 1	1 1 0 1 0 0 0 0		
FWAIT = CPU Wait for NDP	1 0 0 1 1 0 1 1			

FOOTNOTES:

if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if mod = 11 then r/m is treated as an ST(i) field

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP

*except if mod = 000 and r/m = 110 then EA = disp-high: disp-low.

MF = Memory Format
00 — 32-bit Real
01 — 32-bit Integer
10 — 64-bit Real
11 — 16-bit Integer

ST(0) = Current stack top
ST(i) = i^{th} register below stack top

d = Destination
0 — Destination is ST(0)
1 — Destination is ST(i)

P = Pop
0 — No pop
1 — Pop ST(0)

R = Reverse
0 — Destination (op) Source
1 — Source (op) Destination

For **FSQRT:** $-0 \leq ST(0) \leq +\infty$
For **FSCALE:** $-2^{15} \leq ST(1) < +2^{15}$ and ST(1) integer
For **F2XM1:** $0 \leq ST(0) \leq 2^{-1}$
For **FYL2X:** $0 < ST(0) < \infty$
$-\infty < ST(1) < +\infty$
For **FYL2XP1:** $0 < |ST(0)| < (2 - \sqrt{2})/2$
$-\infty < ST(1) < \infty$
For **FPTAN:** $0 \leqslant ST(0) < \pi/4$
For **FPATAN:** $0 \leqslant ST(0) < ST(1) < +\infty$

Bibliographie*

BASIC Reference Manual, IBM (P/N 6025013), Boca Raton, Fl, 1981.

Component Data Catalog, Intel, Santa Clara, CA, 1982.

Disk Operating System Reference Manual, IBM (P/N 6024001), Boca Raton, FL, 1981.

The 8086 Family User's Manual, Intel, Santa Clara, CA, 1980.

The 8086 Family User's Manual Numerics Supplement, Intel, Santa Clara, CA, 1980.

FORTRAN Compiler Reference Manual, IBM (P/N 6024012), Boca Raton, Fl, 1982.

Guide to Operations, IBM (P/N 6025003), Boca Raton, Fl, 1981

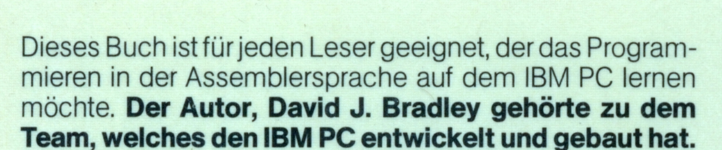
Macro Assembler Reference Manual, IBM (P/N 6024002), Boca Raton, Fl, 1981.

Technical Reference Manual, IBM (P/N 6025008), Boca Raton, Fl, 1981.

* Anmerkung: Sämtliche Handbücher, die von IBM herausgegeben werden, können über autorisierte IBM-PC-Händler bezogen werden.

Register





Dieses Buch ist für jeden Leser geeignet, der das Programmieren in der Assemblersprache auf dem IBM PC lernen möchte. **Der Autor, David J. Bradley gehörte zu dem Team, welches den IBM PC entwickelt und gebaut hat.**

Bradley geht in seinem Buch ausführlich auf Befehle des Intel 8088 in der Assemblersprache und deren Anwendung auf dem IBM Personal Computer sowie auf die Arbeitsweise des IBM Assembler ein. Neben einer Beschreibung des ROM BIOS behandelt der Autor auch Programmieroberflächen für die Hardware des IBM PC und gibt Beispiele für deren Anwendung. Abschließend enthält das letzte Kapitel mehrere Beispiele für den Gebrauch von Assembler-Programmen für die Programmierpraxis auf den IBM Personal Computern.

Das Buch enthält unter anderem:
Beschreibung von binärer Arithmetik und Datendarstellung – Beschreibung des 8088, seiner Register und seiner Adressenmodi – Beschreibung des 8088-Befehlssatzes mit Beispielen für die gebräuchlichsten Befehle – Beschreibung der durch den 8087 zusätzlichen Datentypen und Befehlen – Beschreibung einiger spezieller Programmierhilfen als Teil des Macro-Assemblers – Erläuterung von Hardware und Microcode des IBM PC.



ISBN 3-446-14275-4






HANSER
PHI
Bradley · Programmieren in Assembler
PC professionell